APACHE kafka®

A distributed streaming platform

Archana Jaiswal - corptrainer@rediffmail.com

# USEFUL LINKS

- Jay Kreps links about kafka and streaming in kafka
- https://kafka.apache.org
- Building Realtime data pipeline with Kafka Connect and Spark Streaming b Ewen
- https://www.youtube.com/watch?v=wMLAlJimPzk
- AirStream spark streaming at AIRbnb use case

Archana Jaiswal - corptrainer@rediffmail.com

# MESSAGING SYSTEM

- A Messaging System is responsible for transferring data from one application to another, so the applications can focus on data, but not worry about how to share it.
- Distributed messaging is based on the concept of reliable message queuing.
- Messages are queued asynchronously between client applications and messaging system.
- **Two types** of messaging patterns are available –

❑ point to point

❑ publish-subscribe (pub-sub) messaging system.

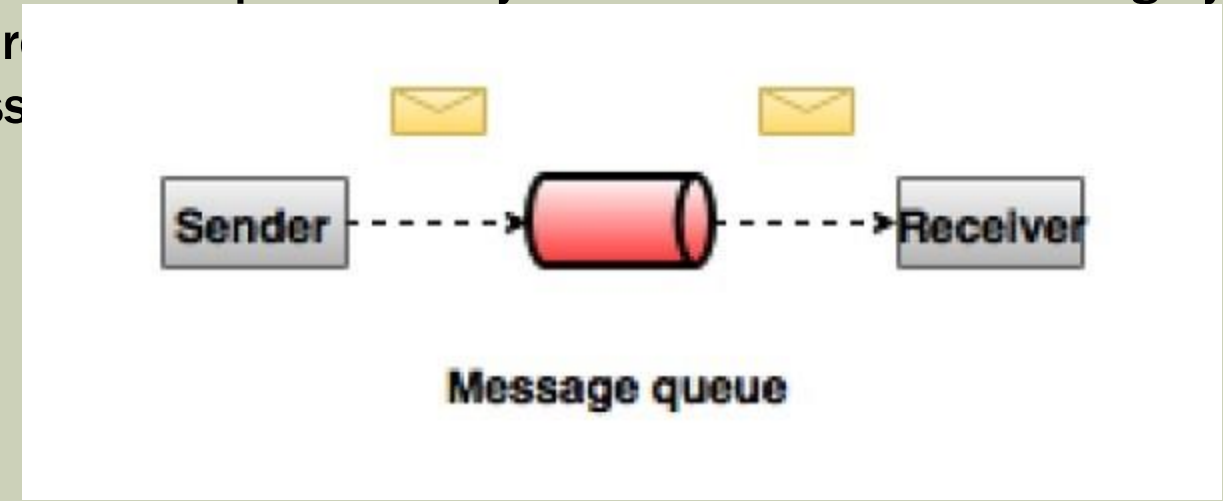
# POINT TO POINT MESSAGING SYSTEM

- In a point-to-point system, messages are persisted in a queue.

One or more consumers can consume the messages in the queue, but a particular message can be consumed by a maximum of one consumer only.

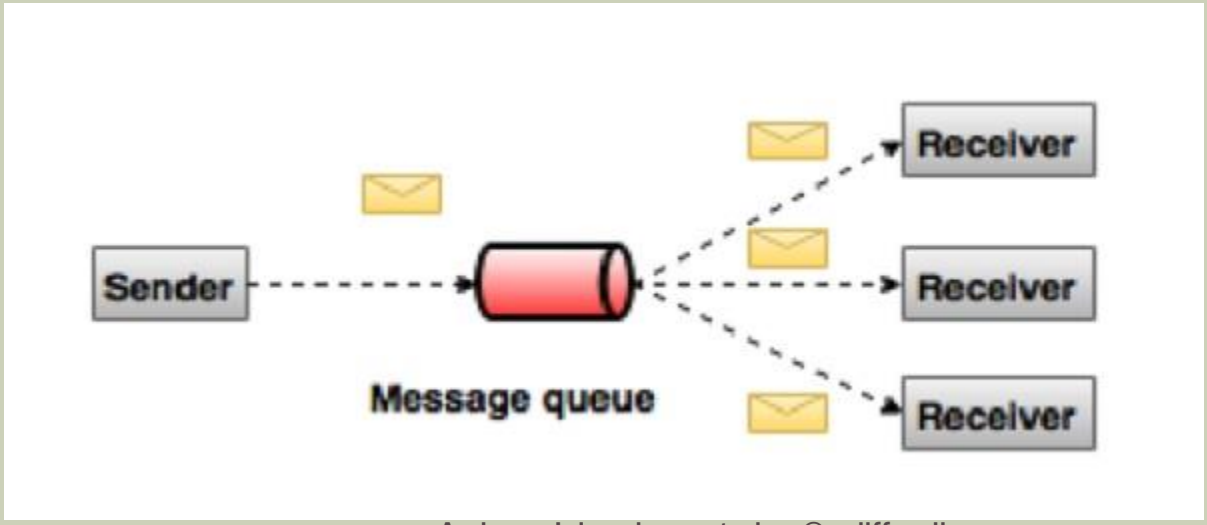
- Once a consumer reads a message in the queue, it disappears from that queue.

The typical example of this system is an Order Processing System, where each or ltiple Order Process



# PUBLISH-SUBSCRIBE MESSAGING SYSTEM

- Messages are persisted in a topic.
- Consumers can subscribe to one or more topic and consume all the messages in that topic.
- Message producers are called publishers and message consumers are called subscribers.
- A real-life example is Dish TV, which publishes different channels like sports, movies, music, etc., and anyone can subscribe to their own set of channels and get them whenever their subscribed channels are available



# INTRODUCTION TO KAFKA

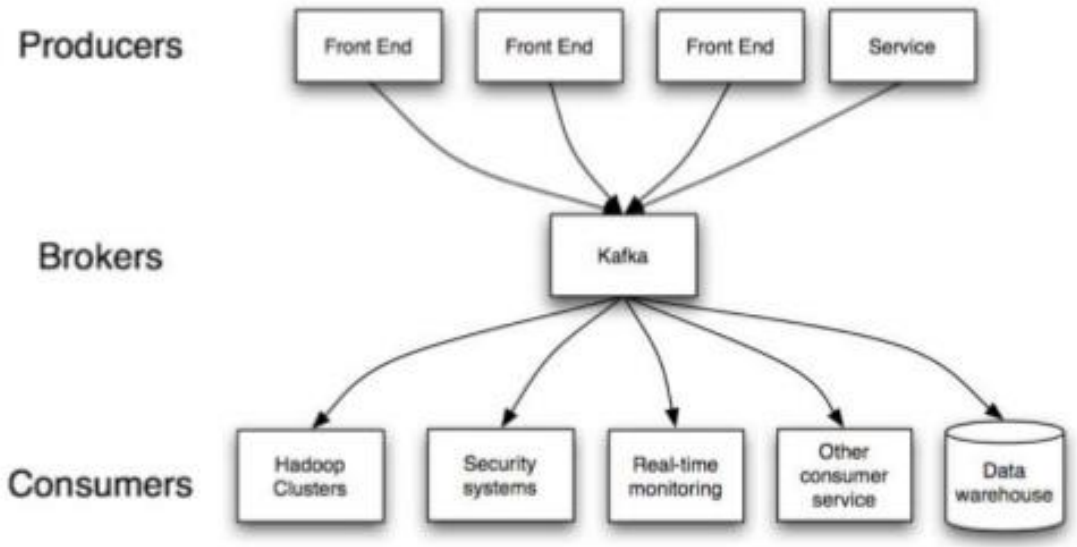
- Apache Kafka is an open-source log Service to build real time pipeline.
- Itis distributed streaming plateform
- It is similar to enterprise messaging system
- Apache Kafka was originated at LinkedIn and later became an open sourced Apache project in 2011, then First-class Apache project in 2012.
- Kafka is written in Scala and Java. Apache Kafka is publish-subscribe based fault tolerant messaging system.
- It is highly tolerable.
- It is fast, scalable and distributed by design

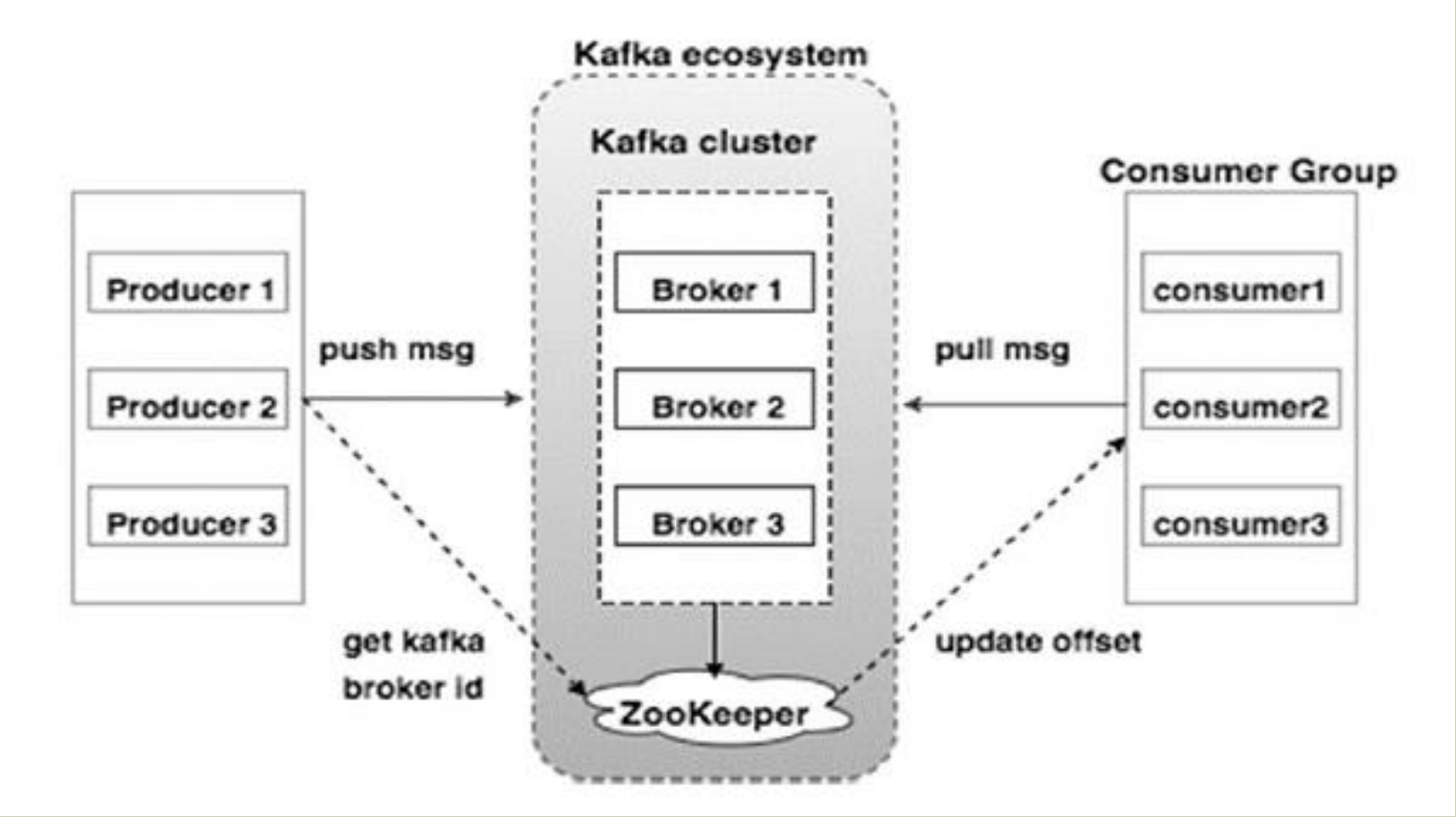
- **Kafka as a Messaging System**
- **Kafka as a Storage System**
- **Kafka for Stream Processing**

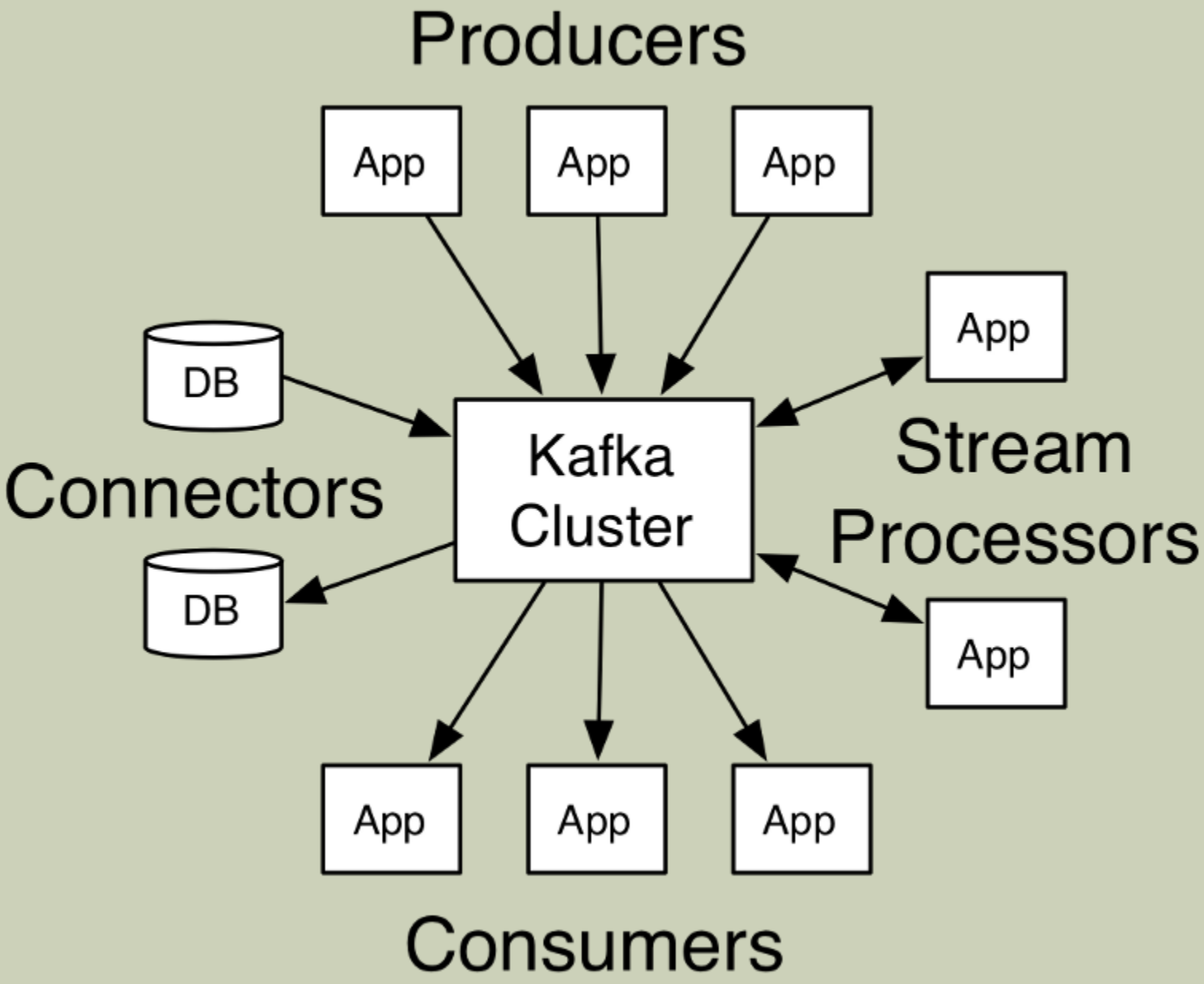
# CHARACTERISTICS OF KAFKA

§ **Scalable**
– Kafka is a distributed system that supports multiple nodes
§ **Fault-tolerant**
– Data is persisted to disk and can be replicated throughout the cluster
§ **High throughput**
– Each broker can process hundreds of thousands of messages per second
§ **Low latency**
– Data is delivered in a fraction of a second
§ **Flexible**
– Decouples the production of data from its consumption

# Kafka decouples data-pipelines




Kafka ecosystem
Kafka cluster
Consumer Group
Producer 1
Producer 2
Producer 3
push msg
Broker 1
Broker 2
Broker 3
pull msg
consumer1
consumer2
consumer3
get kafka
broker id
ZooKeeper
update offset


Producers
App
App
App
App
DB
DB
Kafka
Cluster
Connectors
Stream
Processors
App
App
App
App
App
Consumers

# KAFKA USE CASES

§ **Kafka is used for a variety of use cases, such as**

– Log aggregation

–Messaging

–Web site activity tracking

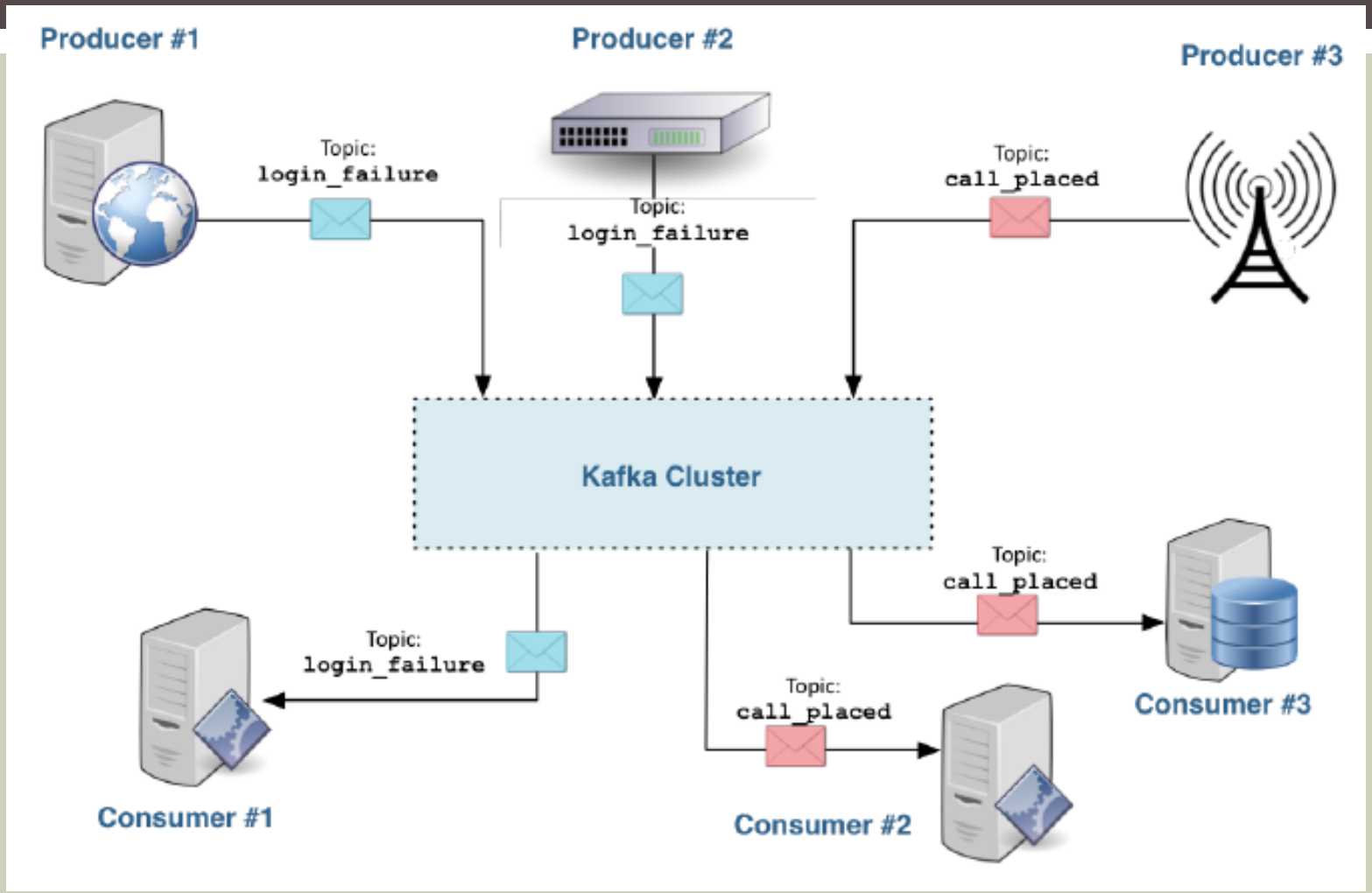
– Stream processing

– Event sourcing

# KAFKA TERMINOLOGY

- Message
- Producer
- Consumer
- Broker
- Topic
- Partition
- Offset
- Consumer Group

# EXAMPLE: HIGH-LEVEL ARCHITECTURE



# MESSAGE

§ **Messages in Kafka are variable-size byte arrays**
– Represent arbitrary user-defined content
– Use any format your application requires
– Common formats include free-form text, JSON

§ **There is no explicit limit on message size**
– Optimal performance at a few KB per message
– Practical limit of 1MB per message

Kafka will retain messages regardless of whether they were read
– Kafka discards messages automatically after the retention period

# MESSAGE

§ **Kafka retains all messages for a defined time period and/or total size**

– Administrators can specify retention on global or per-topic basis

– Kafka will retain messages regardless of whether they were read

– Kafka discards messages automatically after the retention period or
total size is exceeded (whichever limit is reached first)

– Default retention is one week

– Retention can reasonably be one year or longer

NEW ( %ARG ) -
CREATES A
NEW KAFKA::MESSAGE OBJECT. NEW() TAKES AN ARGUMENT - HASH REFERENCE WITH THE MESSAGE ATTRIBUTES CORRESPONDING TO ACCESSORS

PAYLOAD
A SIMPLE MESSAGE RECEIVED FROM THE APACHE KAFKA SERVER.

KEY
THE KEY IS AN OPTIONAL MESSAGE KEY THAT WAS USED FOR PARTITION ASSIGNMENT. THE KEY CAN BE AN EMPTY STRING.

TIMESTAMP
INTEGER OF BIGINT ON 32 BITS PLATFORMS: THE MESSAGE TIMESTAMP ( MIGHT BE -1 IF THE MESSAGE HAS NO TIMESTAMP). REQUIRES KAFKA VERSION > 0.10.0 AND TIMESTAMP ENABLED IN THE TOPIC

**valid**

Boolean value: indicates whether received message is valid or not.

**error**

A description why message is invalid.

**offset**

The offset of the message in the Apache Kafka server.

**next_offset**

The offset of the next message in the Apache Kafka server.

**Attributes**

This holds metadata attributes about the message. The lowest 2 bits contain the compression codec used for the message. The other bits are currently unused.

**HighwaterMarkOffset**

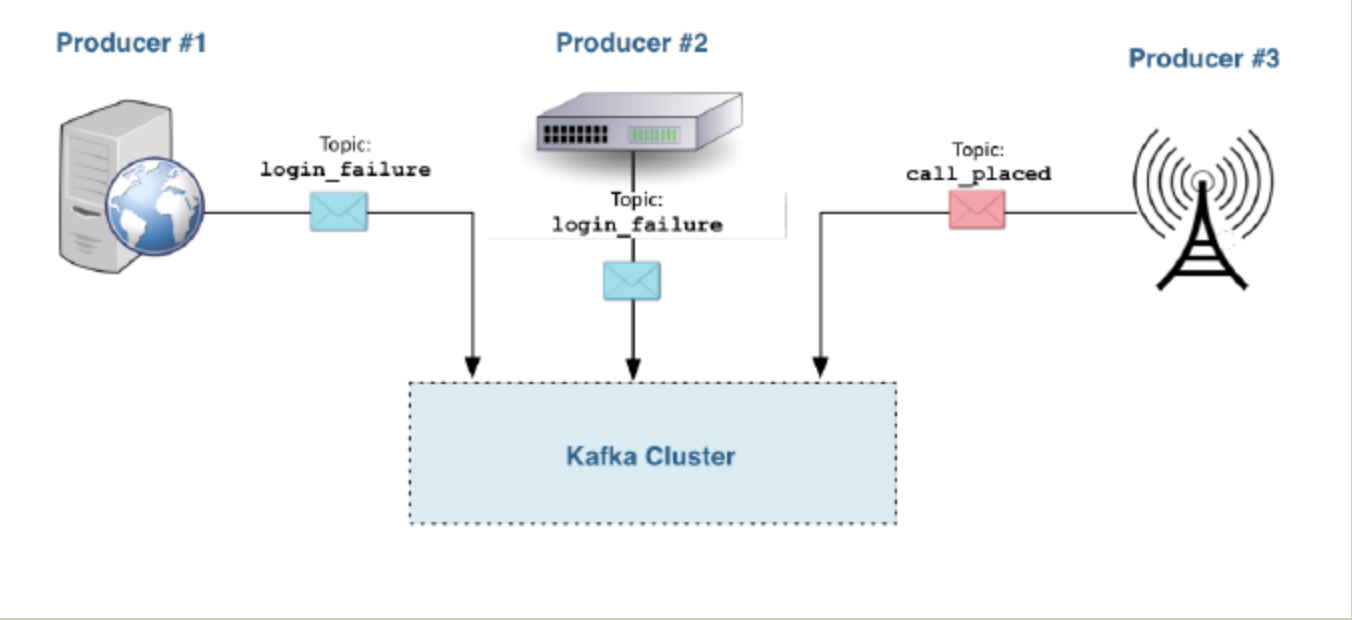
The offset at the end of the log for this partition. This can be used by the client to determine how many messages behind the end of the log they are.

**MagicByte**

This is version id used to allow backwards compatible evolution of the message binary format.

# PRODUCERS

- **Producers publish messages to Kafka topics**
  - They communicate with Kafka, not a consumer
  - Kafka persists messages to disk on receipt



# CONSUMERS

- **A consumer reads messages that were published to Kafka topics**

– They communicate with Kafka, not any producer

- **Consumer actions do not affect other consumers**

– For example, having one consumer display the messages in a topic as they are published does not change what is consumed by other consumers

# KAFKA BROKERS

**Brokers are the fundamental daemons that make up a Kafka cluster or Broker is kafka server**

- **A broker fully stores a topic partition on disk, with caching in memory**
- **A single broker can reasonably host 1000 topic partitions**
- **One broker is elected controller of the cluster**

# CLUSTER

A group of computers sharing workload for a common purpose.

A Kafka Cluster is a group of machine each executing an instance of kafka broker.

As Kafka is Distributed system, each kafka broker is a node in a kafka cluster

# TOPICS

- A topic is a unique name for a kafka stream
- All Kafka messages are organized into *topics*.
- If you wish to send a message you send it to a specific topic and if you wish to read a message you read it from a specific topic.
- A *consumer* pulls messages off of a Kafka topic while *producers* push messages into a Kafka topic
- **There is no explicit limit on the number of topics**
- –However, Kafka works better with a few large topics than many small ones
- **A topic can be created explicitly or simply by publishing to the topic**
- – This behavior is configurable

# PRODUCERS AND CONSUMERS

§ **Tools available as part of Kafka**

– Command-line producer and consumer tools

– Client (producer and consumer) Java APIs

§ **A growing number of other APIs are available from third parties**

– Client libraries in many languages including Python, PHP, C/C++, Go, .NET, and Ruby

§ **Integrations with other tools and projects include**

– Apache Flume

– Apache Spark

–Amazon AWS

– syslog

§ **Kafka also has a large and growing ecosystem**

# SCALING KAFKA

- **Scalability is one of the key benefits of Kafka**

- **Two features let you scale Kafka for performance**

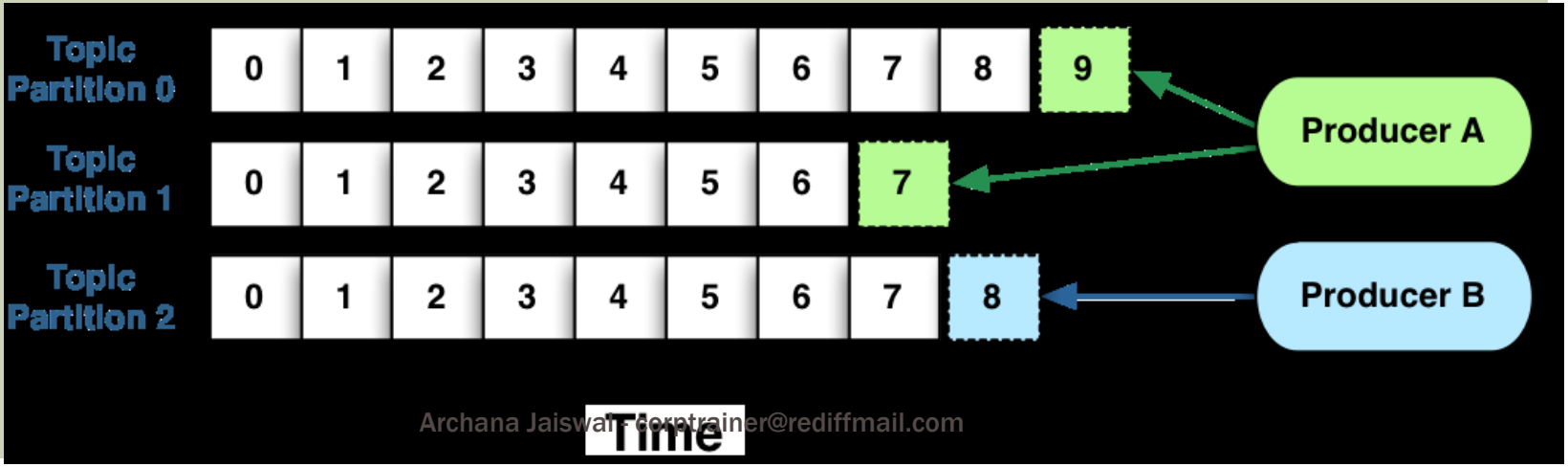
– Topic partitions
– Consumer groups

# KAFKA PARTITION

- Partitions allow you to parallelize a topic by splitting the data in a particular topic across multiple brokers —
- Each partition can be placed on a separate machine to allow for multiple consumers to read from a topic in parallel.
- Consumers can also be parallelized so that multiple consumers can read from multiple partitions in a topic allowing for very high message processing throughput.

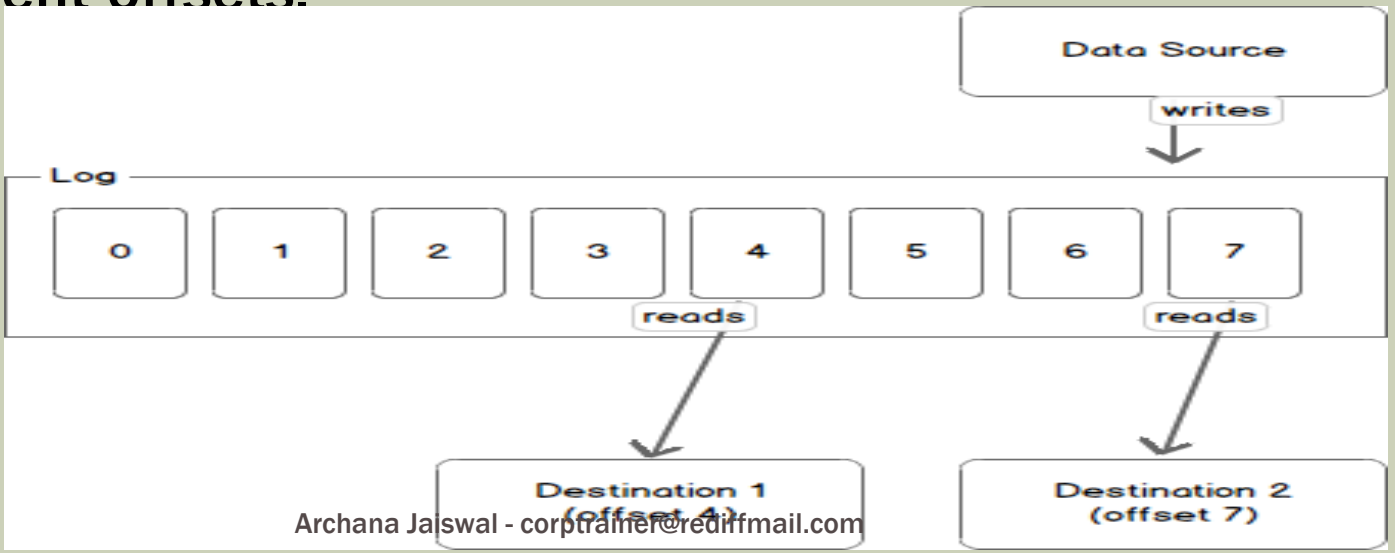
# TOPIC PARTITIONING

- **Kafka divides each topic into some number of partitions ***
- – Topic partitioning improves scalability and throughput
- **A topic partition is an ordered and immutable sequence of messages**
- – New messages are appended to the partition as they are received
- – Each message is assigned a unique sequential ID known as an *offset*



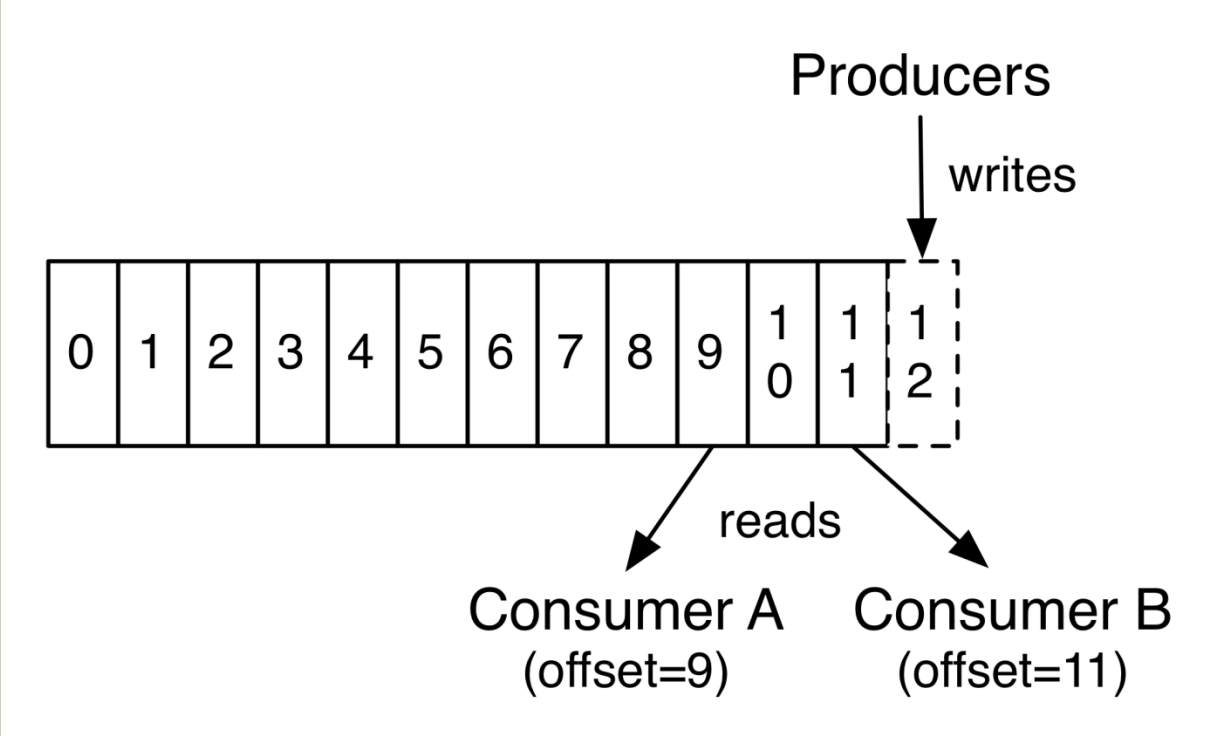
# LOG ANATOMY

- Another way to view a partition is as a log.
- A data source writes messages to the log and one or more consumers reads from the log at the point in time they choose.
- In the diagram below a data source is writing to the log and consumers A and B are reading from the log at different offsets.



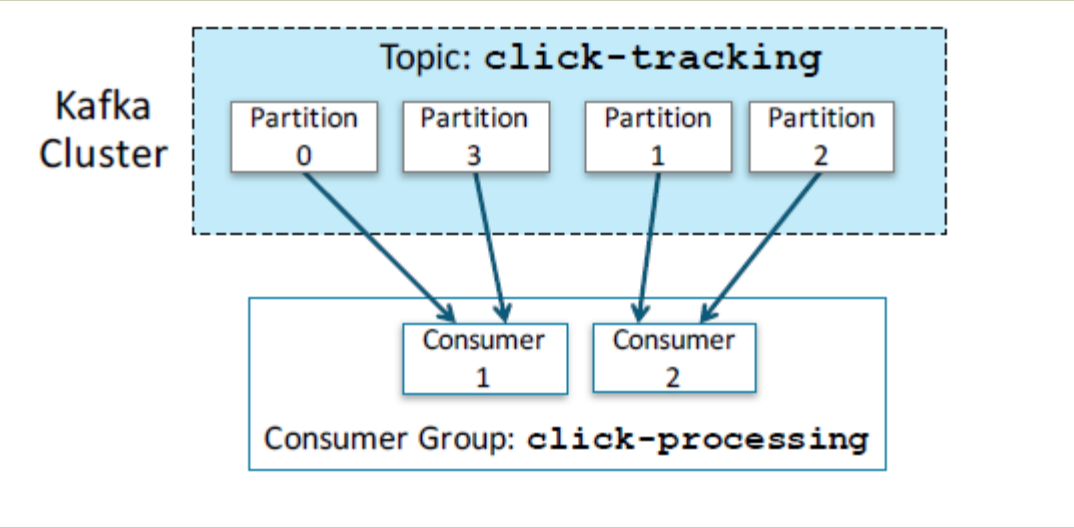
# WHAT IS AN OFFSET

- Each message within a partition has an identifier called its *offset*.
- The offset the ordering of messages as an immutable sequence.
- Kafka maintains this message ordering for you. Consumers can read messages starting from a specific offset and are allowed to read from any offset point they choose, allowing consumers to join the cluster at any point in time they see fit.
- Given these constraints, each specific message in a Kafka cluster can be uniquely identified by a tuple consisting of the message's topic, partition, and offset within the partition.


Producers
writes
0
1
2
3
4
5
6
7
8
9
10
11
12
reads
Consumer A
(offset=9)
Consumer B
(offset=11)

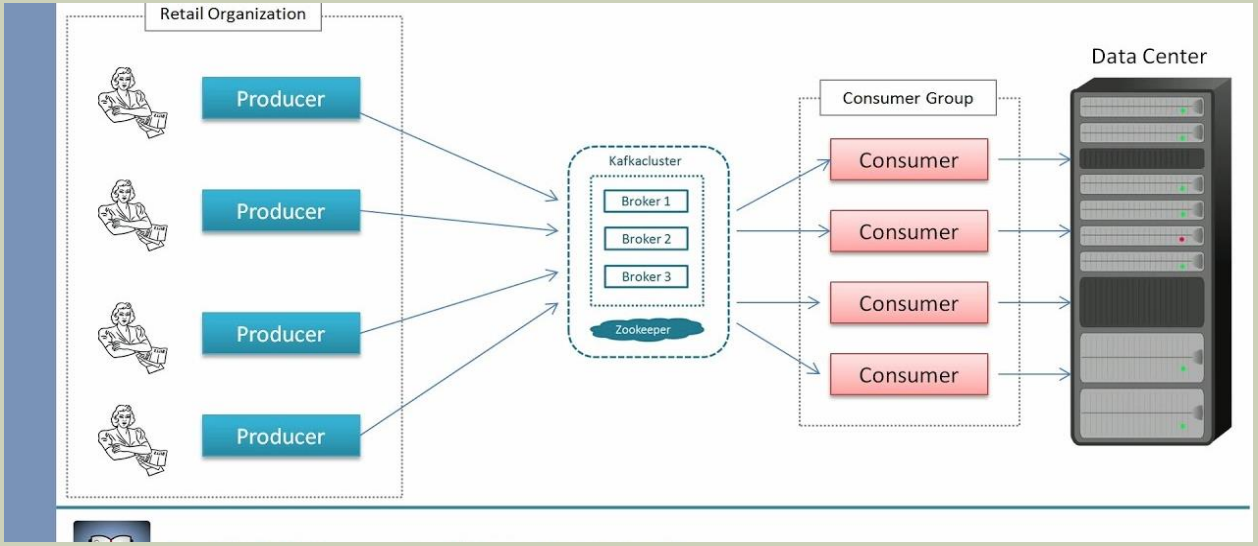
# CONSUMER GROUPS

- One or more consumers can form their own consumer group that work
- together to consume the messages in a topic
- Each partition is consumed by only one member of a consumer group



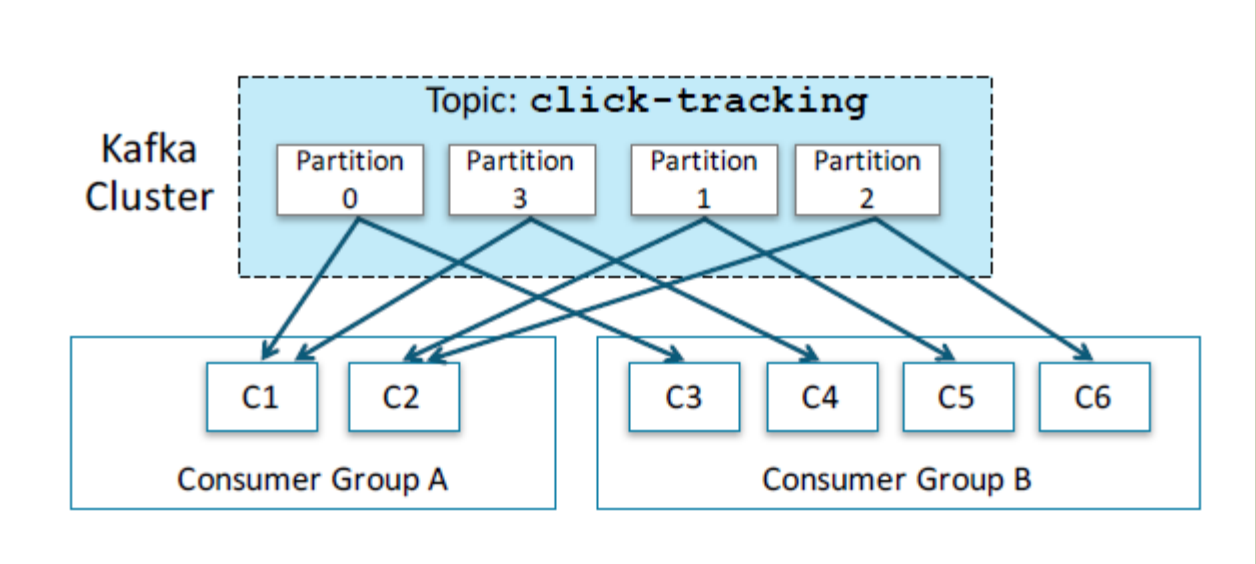
# INCREASING CONSUMER THROUGHPUT

- **Additional consumers can be added to scale consumer group processing**
- **Consumer instances that belong to the same consumer group can be in separate processes or on separate machines**



# MULTIPLE MESAAGE GROUPS

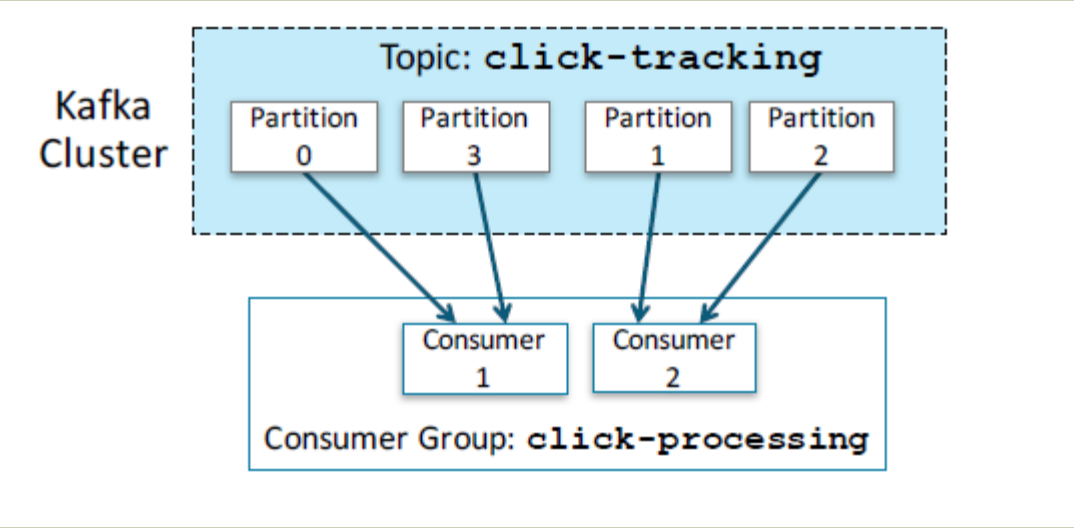
- **Each message published to a topic is delivered to one consumer instance within each subscribing consumer group**



# CONSUMER GROUPS

**One or more consumers can form their own consumer group that work together to consume the messages in a topic**

**§ Each partition is consumed by only one member of a consumer group**



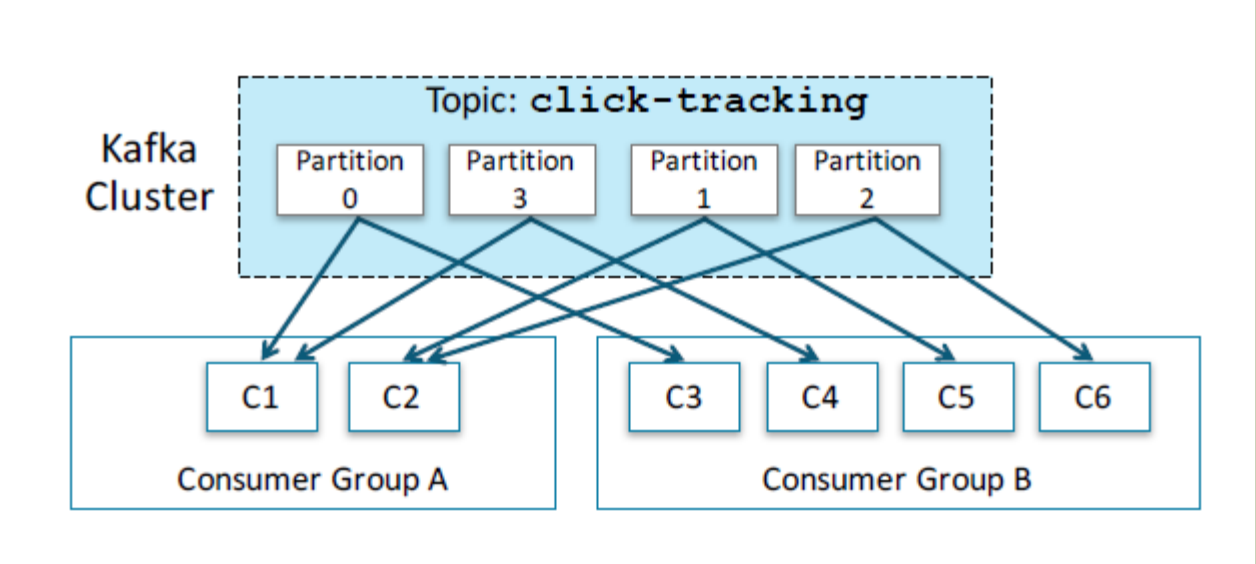
# INCREASING CONSUMER THROUGHPUT

§ **Additional consumers can be added to scale consumer group processing**

§ **Consumer instances that belong to the same consumer group can be in separate processes or on separate machines**

# MULTIPLE MESAAGE GROUPS

§ Each message published to a topic is delivered to one consumer instance within each subscribing consumer group



# START KAFKA

- Download kafka
- tar –xzf kafka.....tgz
- Start zookeeper
- Bin/zookeeper-server-start.sh config/zookeeper.properties
- Check port no – 2181(default)
- bin/kafka-server-start.sh config/server.properties

- Create a topic
- Bin/kafka-topics.sh –zookeeper localhost:/2181 –create –topic Mytopic –partitions 2 –replication-factor 1
- Run Producer
- bin/kafka-console-producer.sh –broker-list localhost:9092 –topic Mtopic
- Run Consumer
- Bin/kafka-console-consumer.sh –bootstrap-server localhost:9092 –topic Mytopic

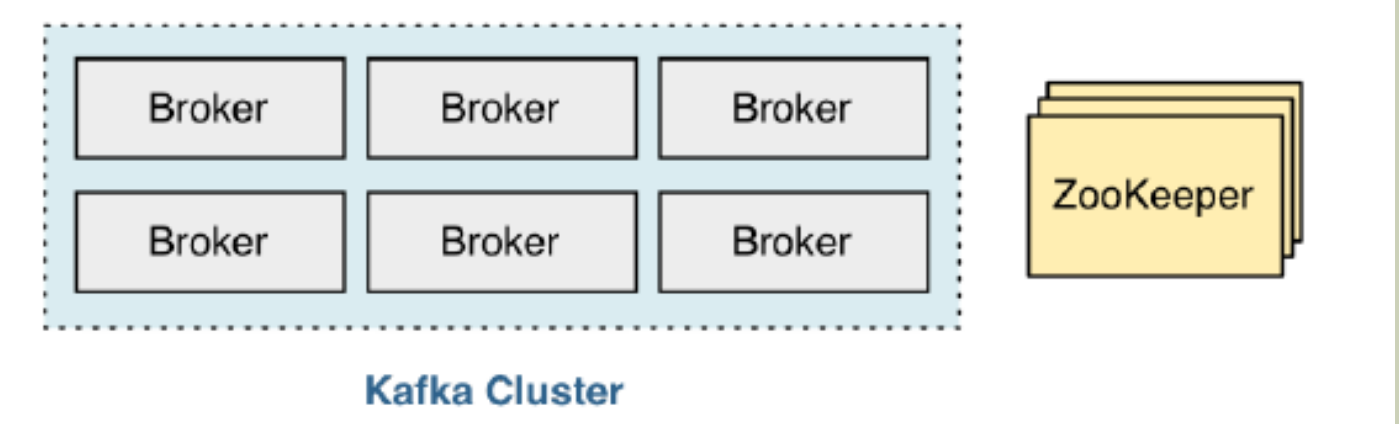
# PUBLISH AND SUBSCRIBE TO TOPIC

**Kafka functions like a traditional queue when all consumer instances**
**belong to the same consumer group**
– In this case, a given message is received by one consumer
§ **Kafka functions like traditional publish-subscribe when each consumer**
**instance belongs to a different consumer group**
– In this case, all messages are broadcast to all consumer groups

# KAFKA CLUSTERS

§ A Kafka cluster consists of one or more *brokers*—servers running the Kafka broker daemon

§ Kafka depends on the Apache ZooKeeper service for coordination

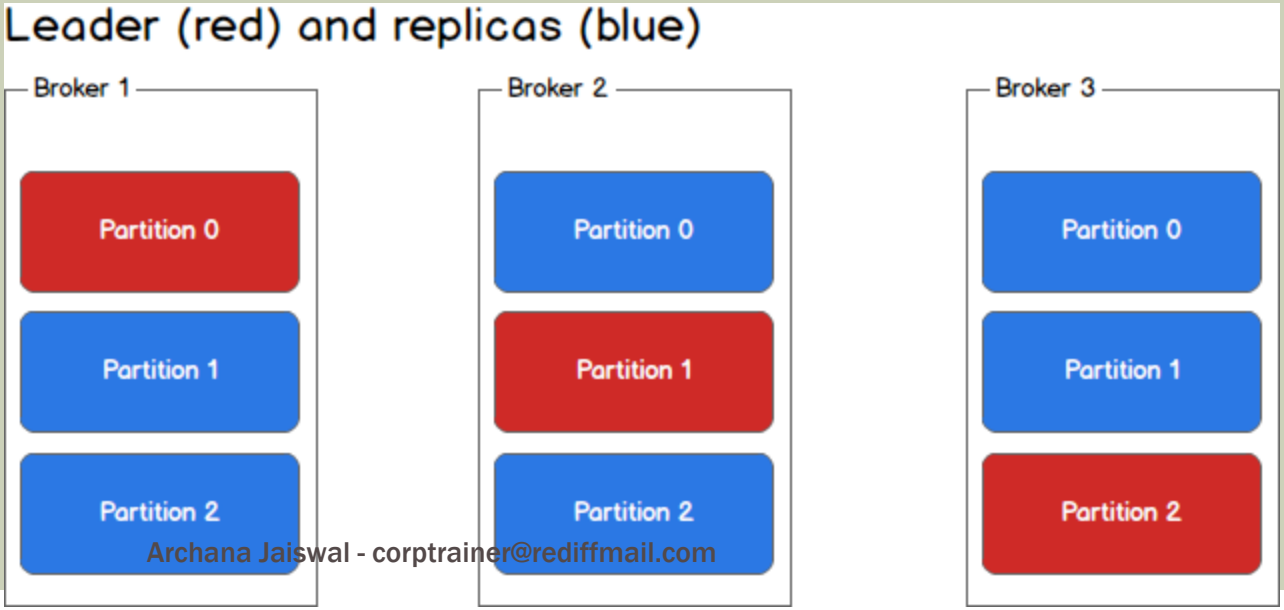


# APACHE ZOOKEEPER

§ Apache ZooKeeper is a coordination service for distributed applications
§ Kafka depends on the ZooKeeper service for coordination
§ Kafka uses ZooKeeper to keep track of brokers running in the cluster
§ Kafka uses ZooKeeper to detect the addition or removal of consumers

# FAULT TOLERANCE

- Each broker holds a number of partitions and each of these partitions can be either a leader or a replica for a topic.
- All writes and reads to a topic go through the leader and the leader coordinates updating replicas with new data.
- If a leader fails, a replica takes over as the new leader.



# MESSAGES ARE REPLICATED

**Configure the producer with a list of one or more brokers**

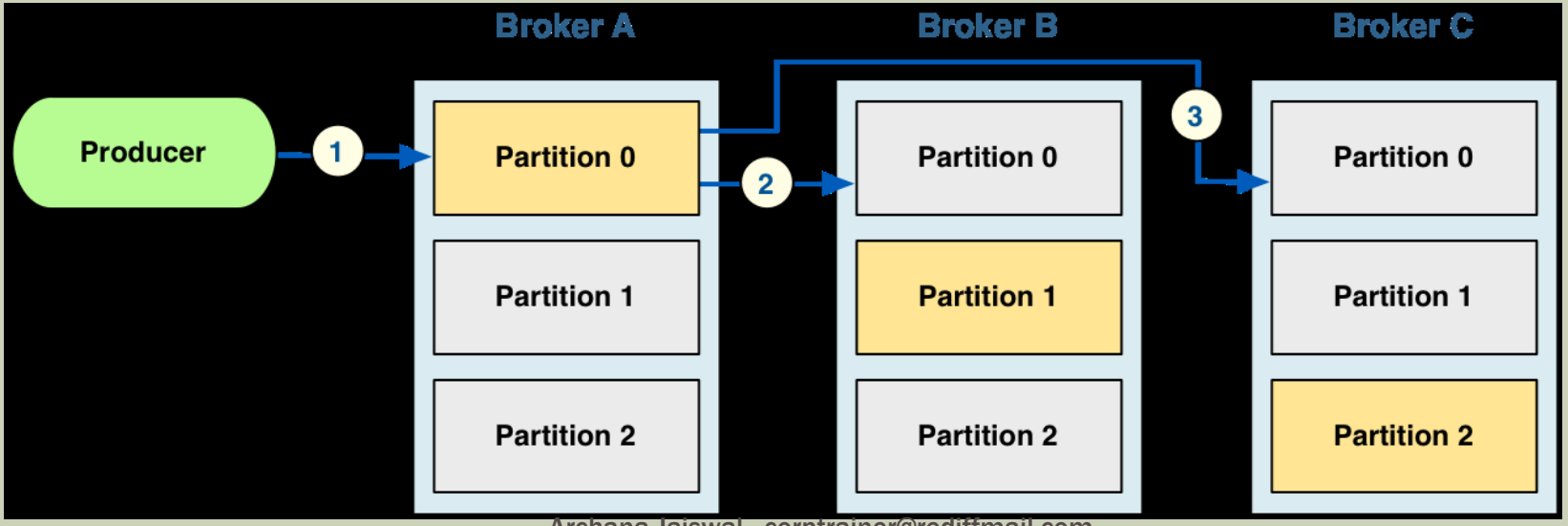
– The producer asks the first available broker for the leader of the desired topic partition

**The producer then sends the message to the leader**

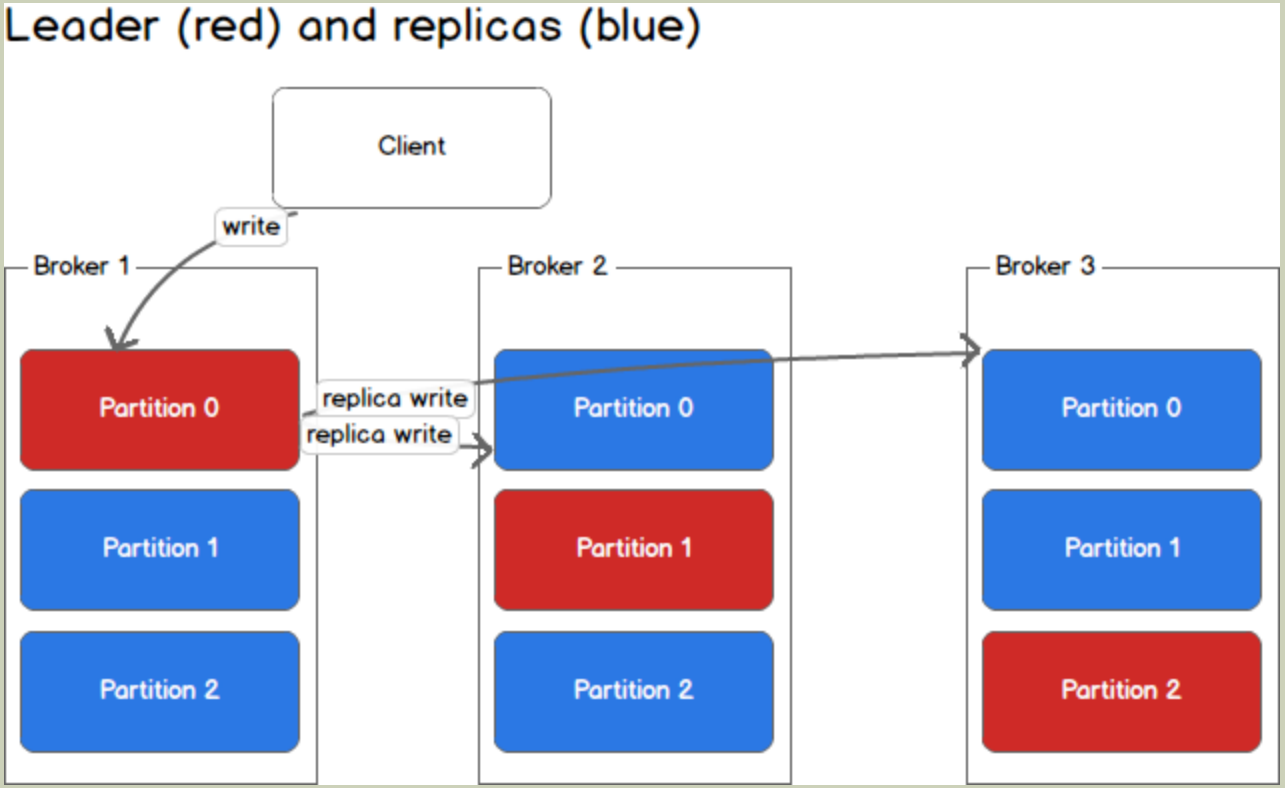
– The leader writes the message to its local log

– Each follower then writes the message to its own log

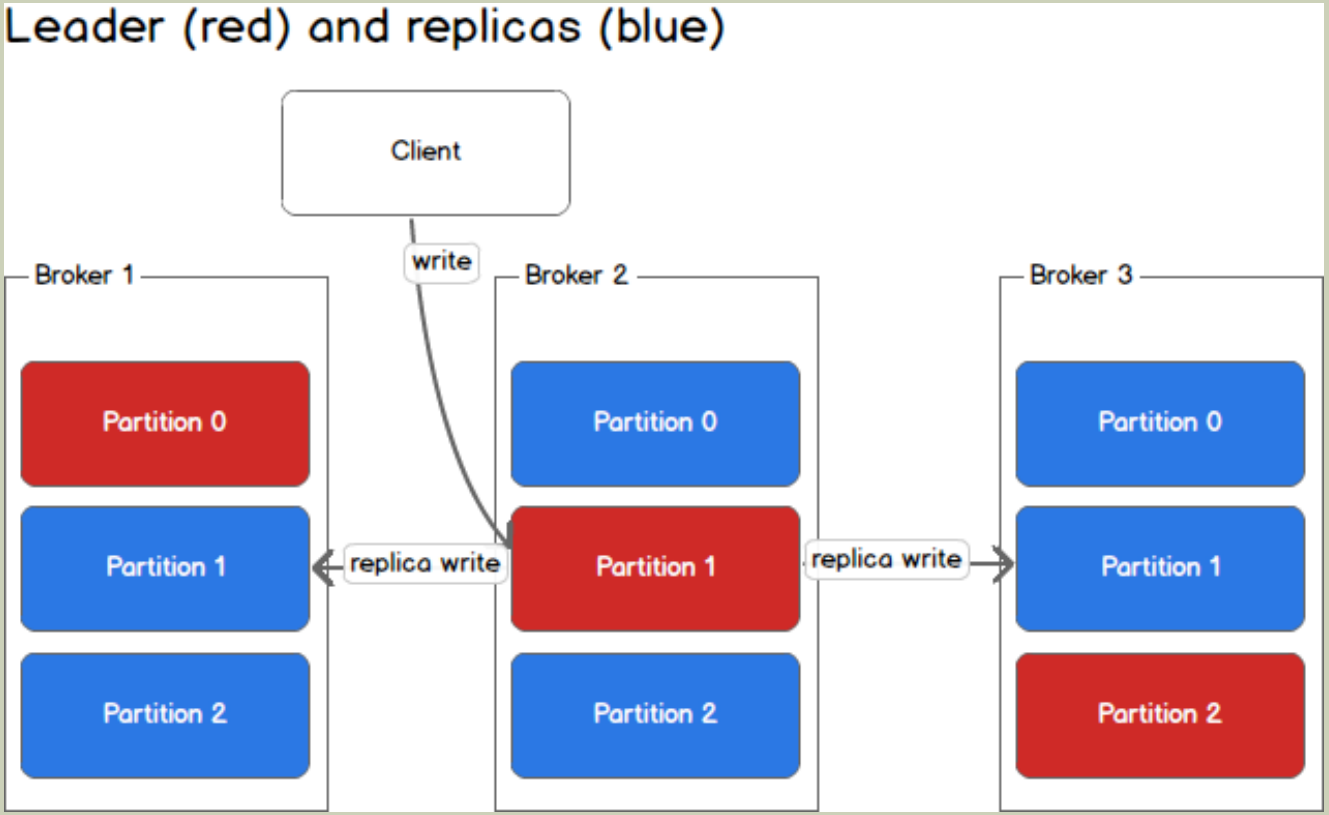
– After acknowledgements from followers, the message is committed



- Producers write to a single leader, this provides a means of load balancing production so that each write can be serviced by a separate broker and machine.
- In the first image, the producer is writing to partition 0 of the topic and partition 0 replicates that write to the available replicas.

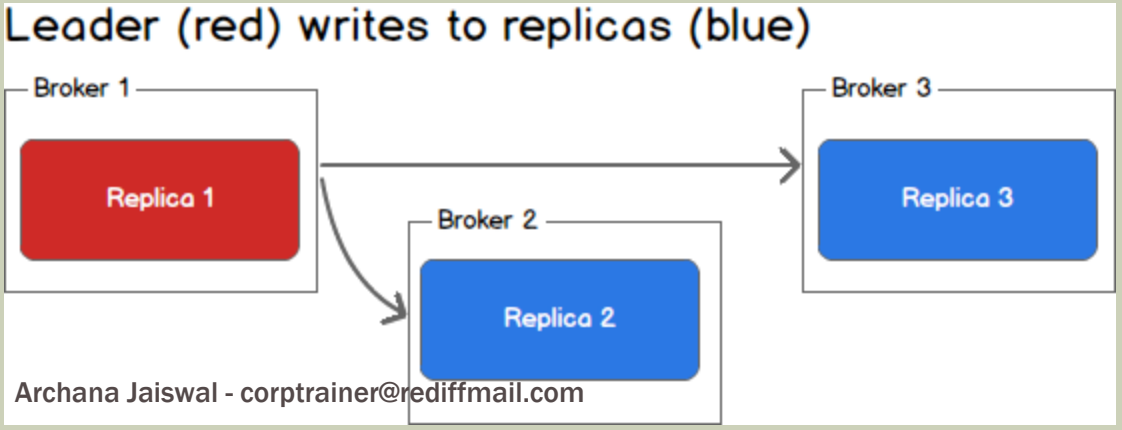


- In the second image, the producer is writing to partition 1 of the topic and partition 1 replicates that write to the available replicas.
- Since each machine is responsible for each write, throughput of the system as a whole is increased.



# HANDLING WRITES

- When communicating with a Kafka cluster, all messages are sent to the partition's leader.
- The leader is responsible for writing the message to its own in sync replica and, once that message has been committed, is responsible for propagating the message to additional replicas on different brokers.
- Each replica acknowledges that they have received the message and can now be called in sync.



# LEADER WRITES TO REPLICAS

When every broker in the cluster is available, consumers and producers can happily read and write from the leading partition of a topic without issue.

# CREATING TOPICS FROM THE COMMAND LINE

- **Kafka includes a convenient set of command line tools**
- – These are helpful for exploring and experimentation
- **The kafka-topics command offers a simple way to create Kafka topics**

– Provide the topic name of your choice, such as **device_status**

```
kafka-topics –create \
–zookeeper zkhost1:2181,zkhost2:2181,zkhost3:2181 \
–replication-factor 3 \
–partitions 5 \
–topic device_status
```

# RUNNING A PRODUCER FROM THE COMMAND LINE

**You can run a producer using the kafka-console-producer tool**

- **Specify one or more brokers in the –broker-list option**

– Each broker consists of a hostname, a colon, and a port number

– If specifying multiple brokers, separate them with commas

```
kafka-console-producer \
–broker-list brokerhost1:9092,brokerhost2:9092 \
–topic device_status
```

# RUNNING A CONSUMER FROM THE COMMAND LINE

**You can run a consumer with the kafka-console-consumer tool**

- **This requires the ZooKeeper connection string for your cluster**

–Unlike starting a producer, which instead requires a list of brokers

- **The command also requires a topic name**

```
kafka-console-consumer \
–zookeeper
zkhost1:2181,zkhost2:2181,zkhost3:2181 \
–topic device_status \
–from-beginning
```

# BROKER CONFIGURATION

- Broker.id
- Port
- Log.dirs
- Zookeeper.connect
- Delete.topic.enable
- Auto.create.topics.enable
- Default.replication.factor
- Log.retention.ms
- Log.retention.byte

# PRODUCER API

# CONSUMER API

# CALLBACK AND ACKNOWLEDGMENT

- **Fire and forget**
- **Synchronous Send**
- **Asynchronus Send/callback**

# DEFAULT PARTITIONER

1. If a partition is specifies in the record, use it
2. If no partition is specified but a key is present choose a partition based on a hash of the key
3. If no partition key is present choose  a partition in a round robin fashion

# CUSTOM PARTITIONER