Kafka Component

WL

Outline

- Kafka Components
- Kafka Ecosystems

Introduction

Kafka Outline

- Introduction
- Kafka Components
- Kafka Ecosystems

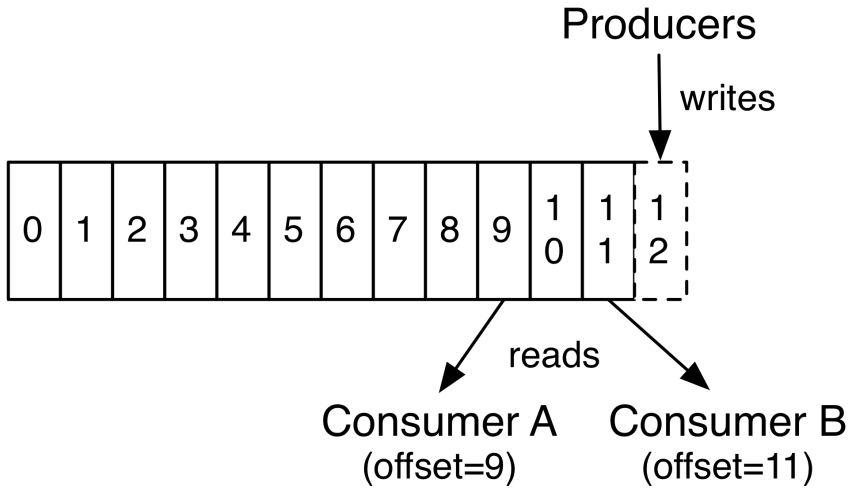
Why Using Kafka

- 以時間複雜度為 O(1) 的方式提供消息持久化能力，對 TB 級以上數據也能保證常數時間複雜度的訪問性能。
- 高吞吐率，同時保證每個 Partition 內的消息順序傳輸。
- 同時支持離線數據處理和即時數據處理。
- Scale out：支持在線水平擴展。

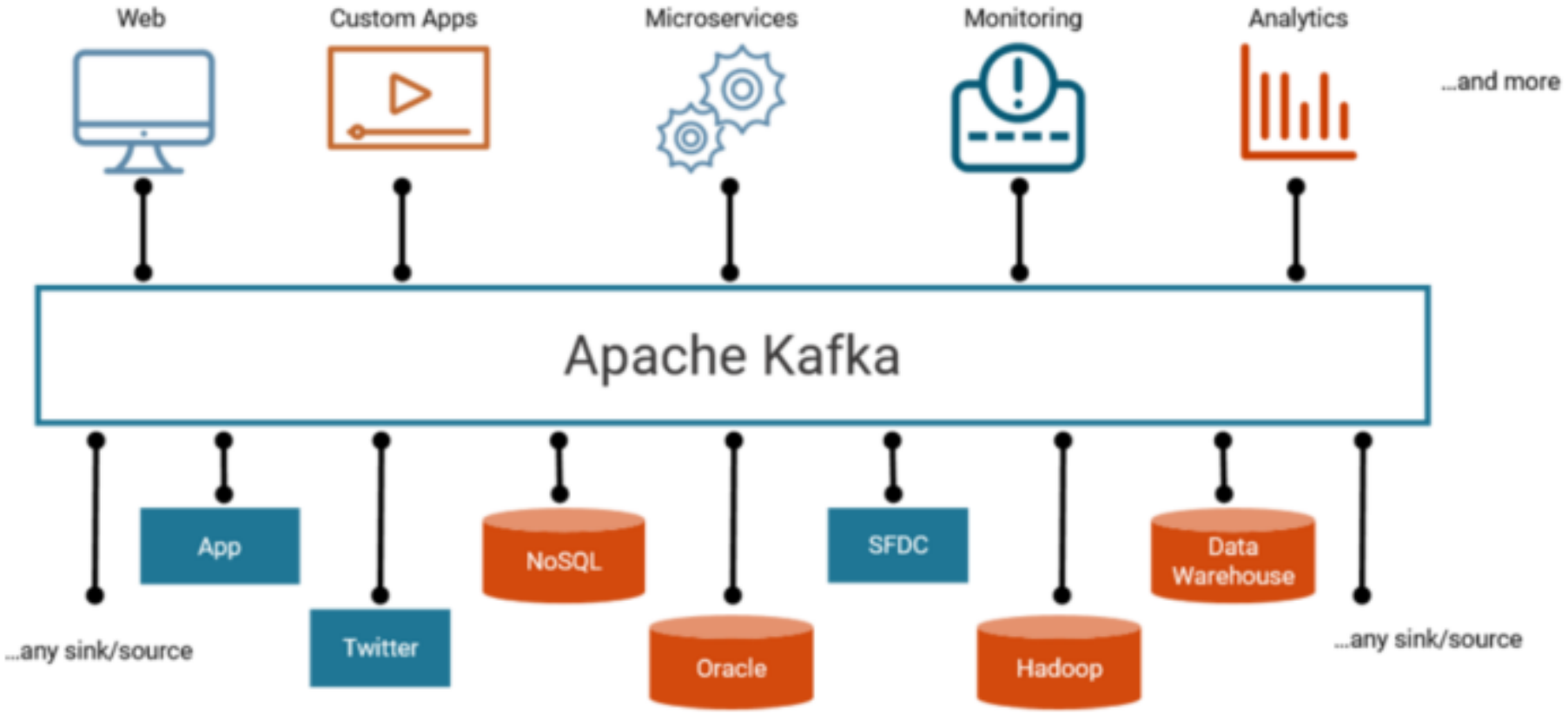
Why Kafka is Fast

ATHEMASTER

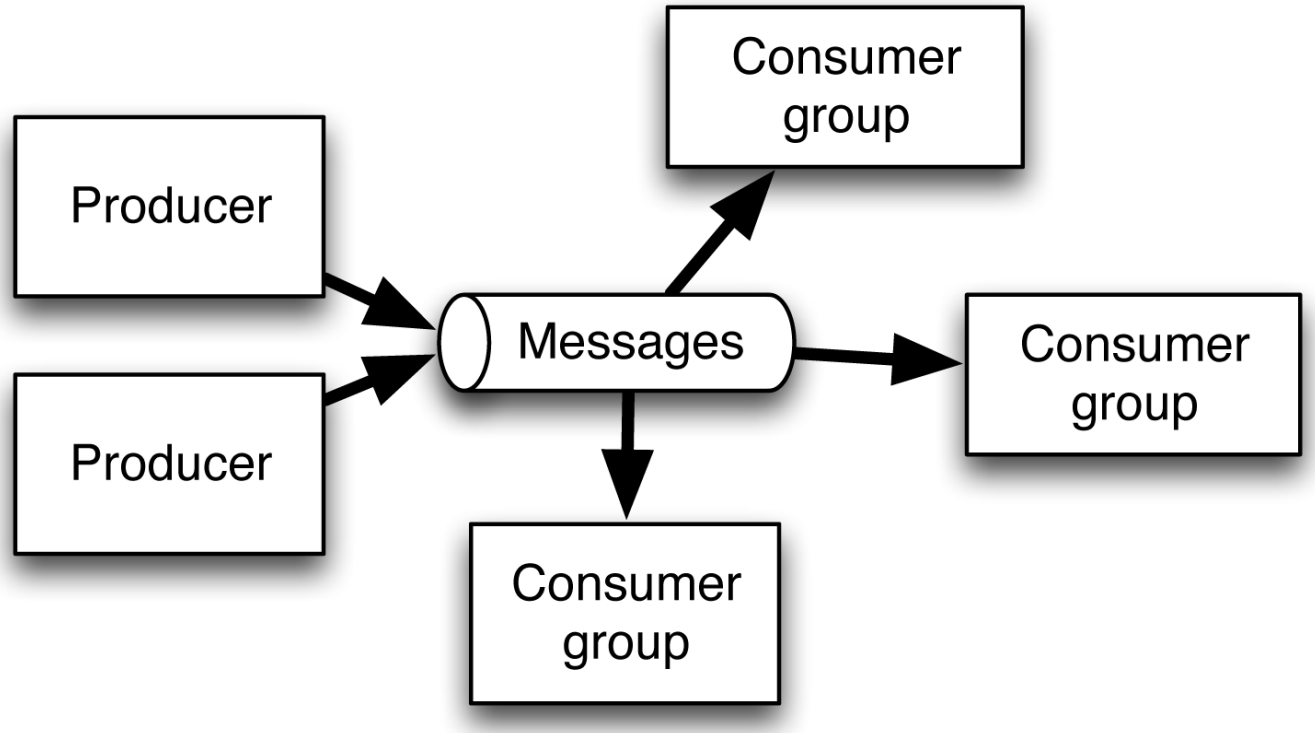
- Kafka 為一個 immutable 數據結構， producer 僅可在數據結構的末尾 append 條目，不進行 update 或 delete 現有數據， consumer 亦對連續存放的資料進行讀取



Decoupling of data streams



Message Passing Infrastructure



Kafka CLI

Kafka CLI

- start kafka server
  - cd ~/workspace/kafka/kafka_2.11-2.0.0
  - bin/kafka-server-start.sh config/server.properties
- list topics
  - bin/kafka-topics.sh --list --zookeeper localhost:2181

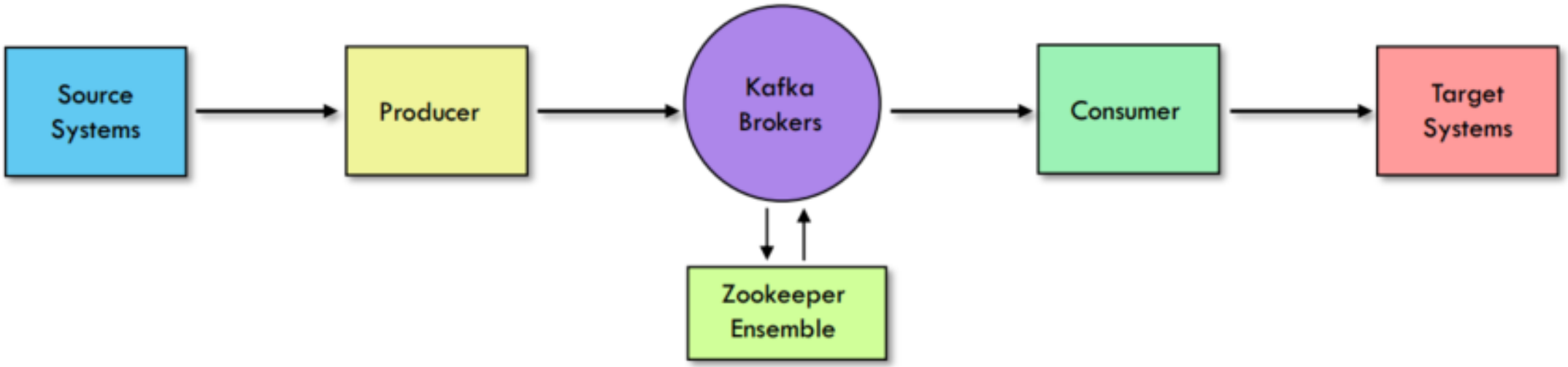
Kafka CLI

- create topic
  - bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 3 --topic test
  - bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic test
- delete topic
  - bin/kafka-topics.sh --delete --zookeeper localhost:2181 --topic test

Kafka CLI

- producer
  - bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
- consumer
  - bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --from-beginning

Kafka Architecture



Kafka Components

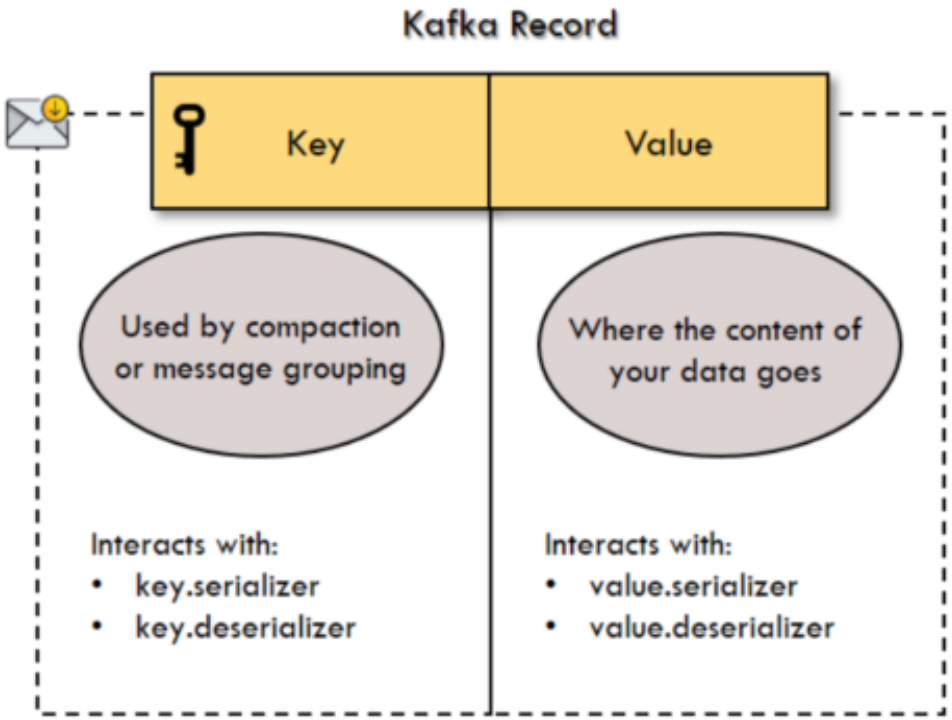
Components

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

Message

ATHEMASTER

- Message 是通信的基本單位，每個訊息在 Kafka 中稱為 Record，每個 Record 包含
  - key : 訊息的鍵值，可用在 grouping 和 compaction
  - value : 訊息的內容



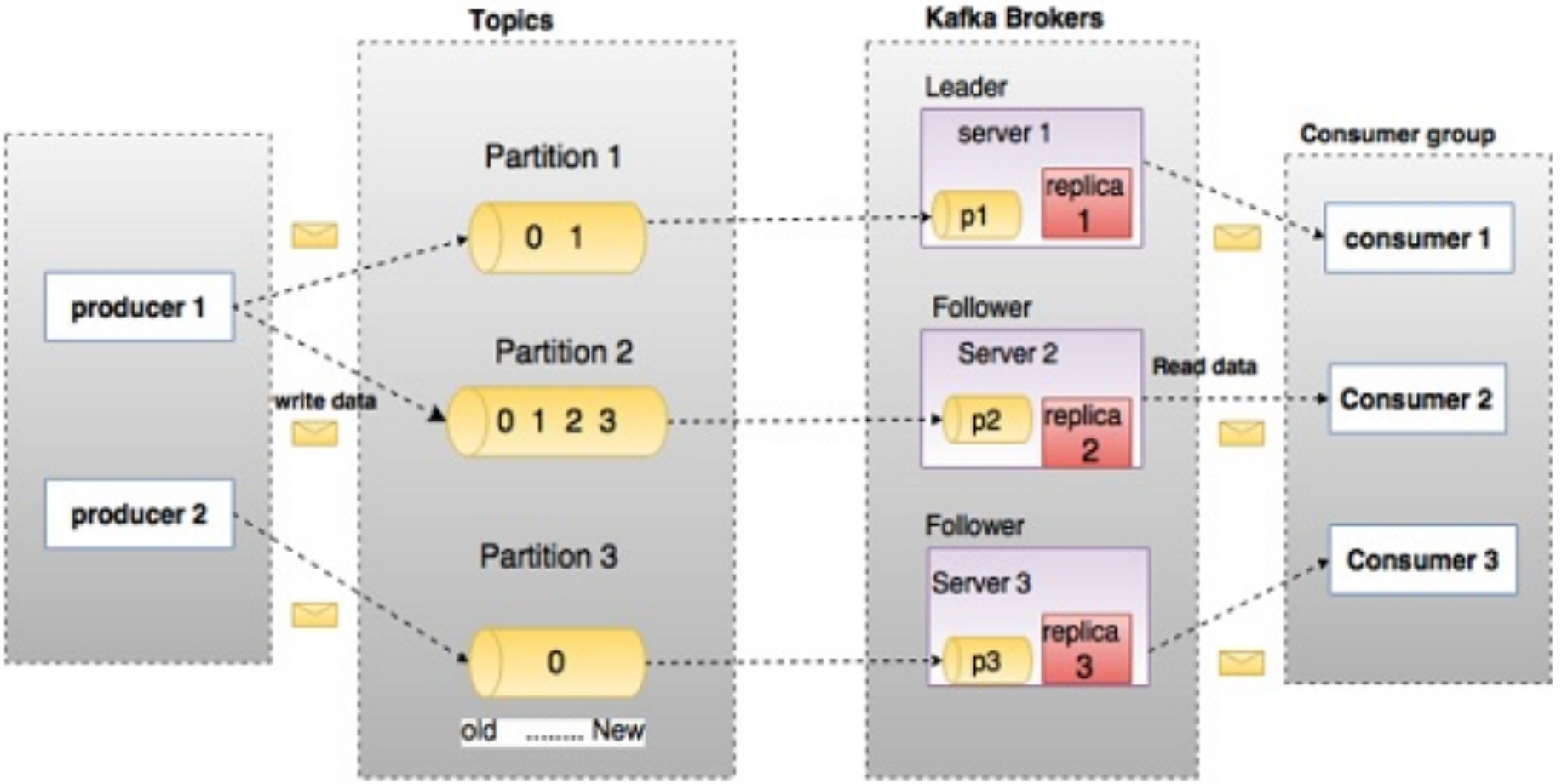
Topic

- 每條發送到 Kafka cluster 的 message 都屬於某個類別，這個類別就被稱為 topic。
  - Like a table in a database
  - Kafka cluster 可建立多個 topics
- 為了實現可擴展性，topic 可被切分為多個部分，這個部分在 kafka 內被稱為 partition

Partition

- Partition 是 Topic 物理上的分組，一個 topic 可以分為多個 partition，每個 partition 是一個有序的隊列
  - 可以保證一個 partition 中的訊息，按照順序發送給 consumer
- Partition 是平行處理的最小單位，換句話說，一個 partition 同一時間只可被 consumer group 中的一個 consumer 處理

Topic & Partitions



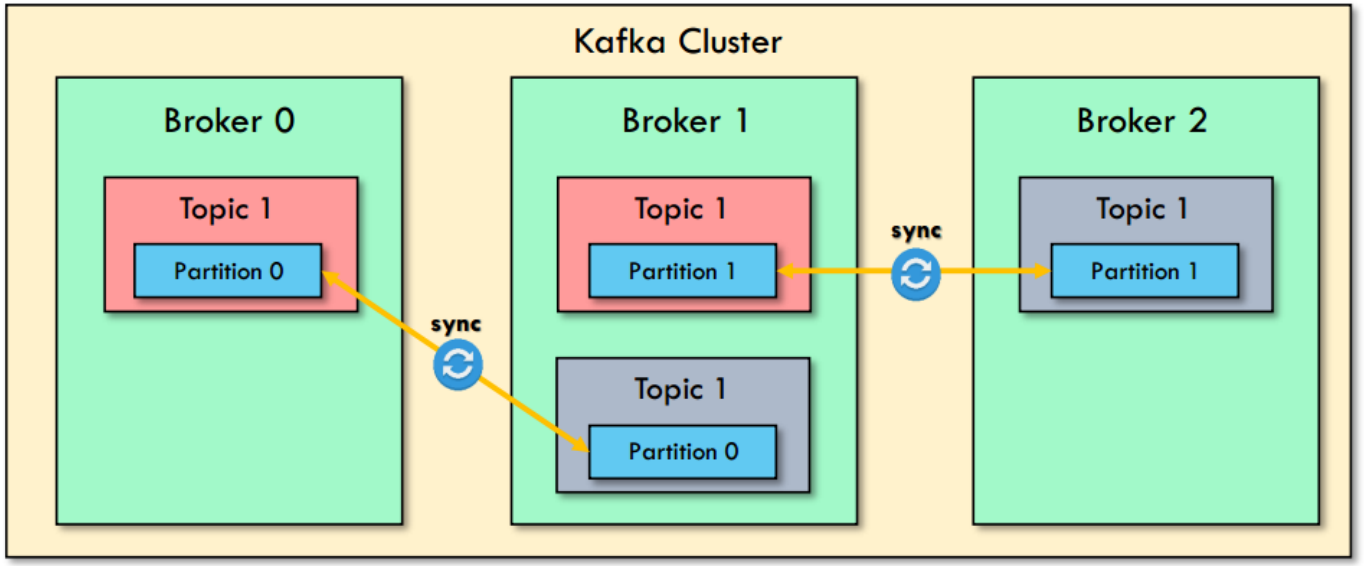
Broker

- 一個 Kafka service 即是一個 broker，而多個 broker 可組成一個 Kafka cluster，broker 和 broker 之間沒有 master 和 slaver 的差別，他們之間地位是平等的
- 當連線到 cluster 中的某台 broker，即可和 broker cluster 中所有 broker 的資訊

Replica

ATHEMASTER

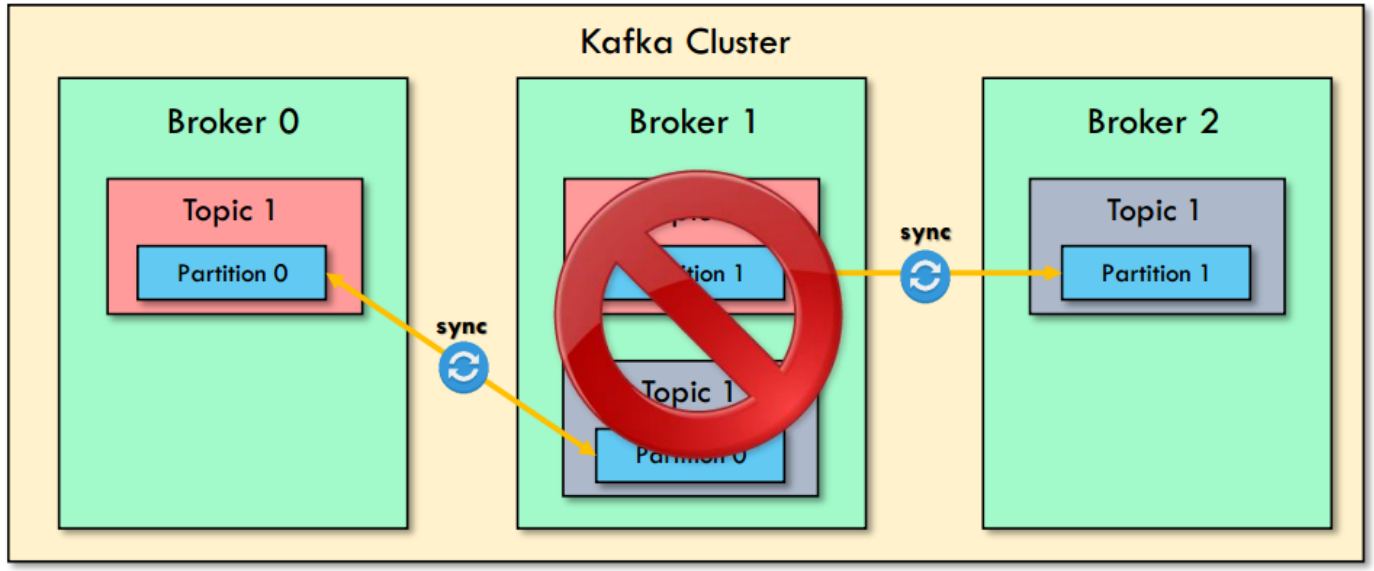
- Replica 即副本，每個 topic 中的 partition，可以建立 > 1 的副本資料，當 broker 異常終止時，可以取代原有的服務讓系統持續運作



Replica

ATHEMASTER

- 當 broker 1 終止時， broker 0 保有它原有的服務，broker 2 則接替 broker 1 原有提供的服務

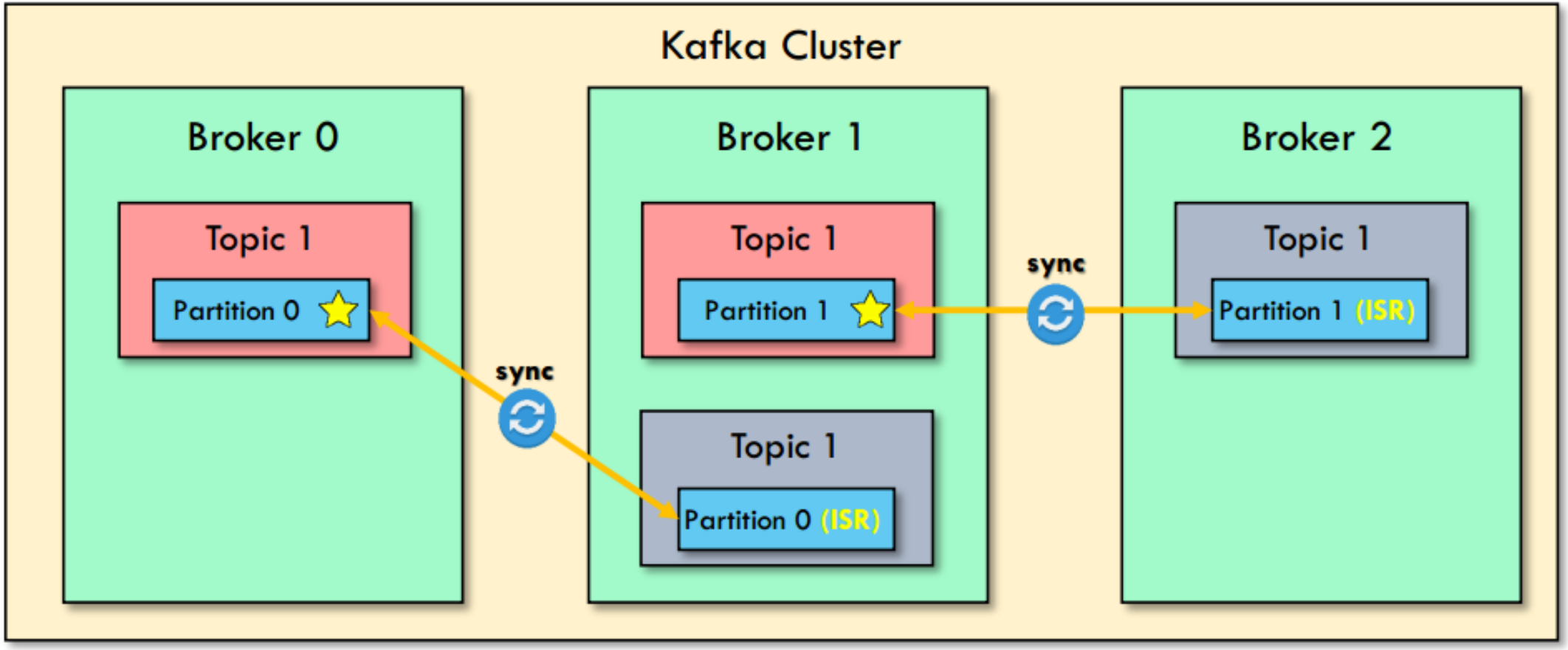


Leader for a partition

- 同一時間，一個 partition 只會有一個 broker 可以當作其 leader，只有當作 leader 的 partition 可以提供資料給 consumer 使用
- 其他 broker 扮演 follower 或 ISR (in-sync replica) 的角色

Leader for a partition

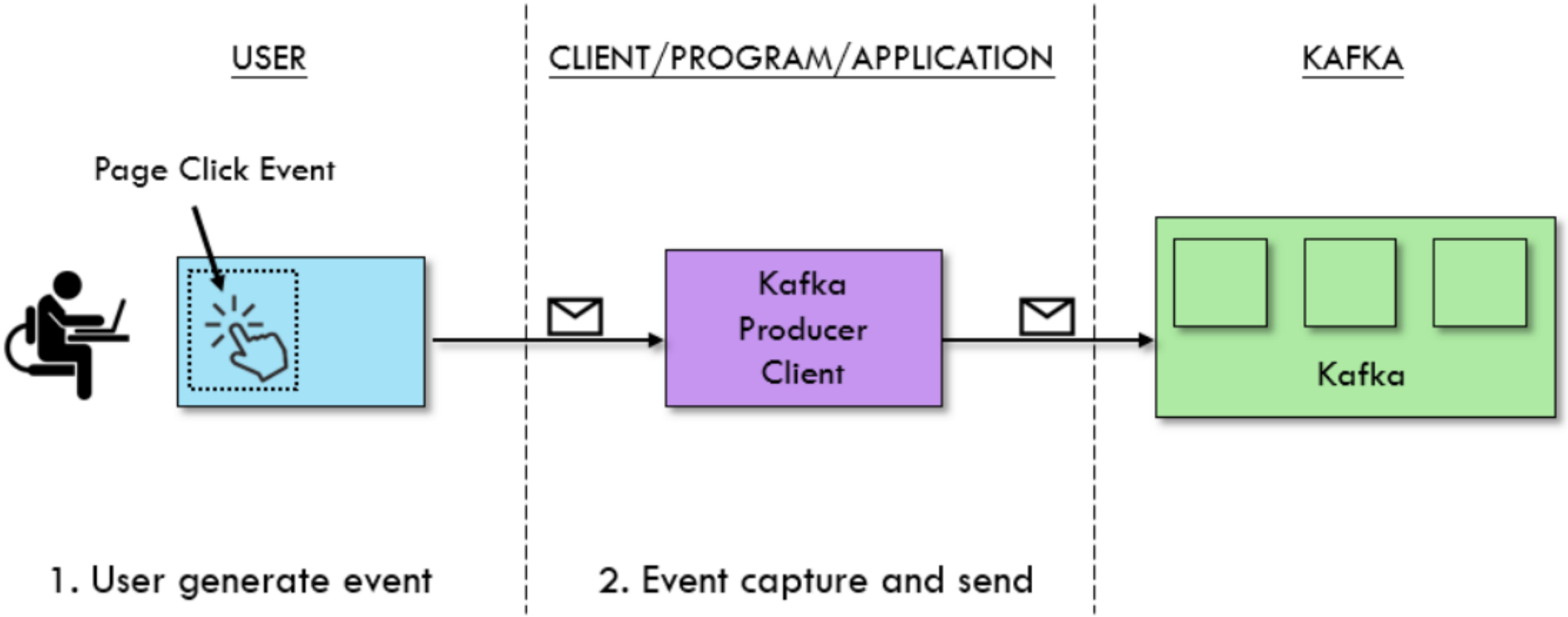
ATHEMASTER



Broker Leader Elections

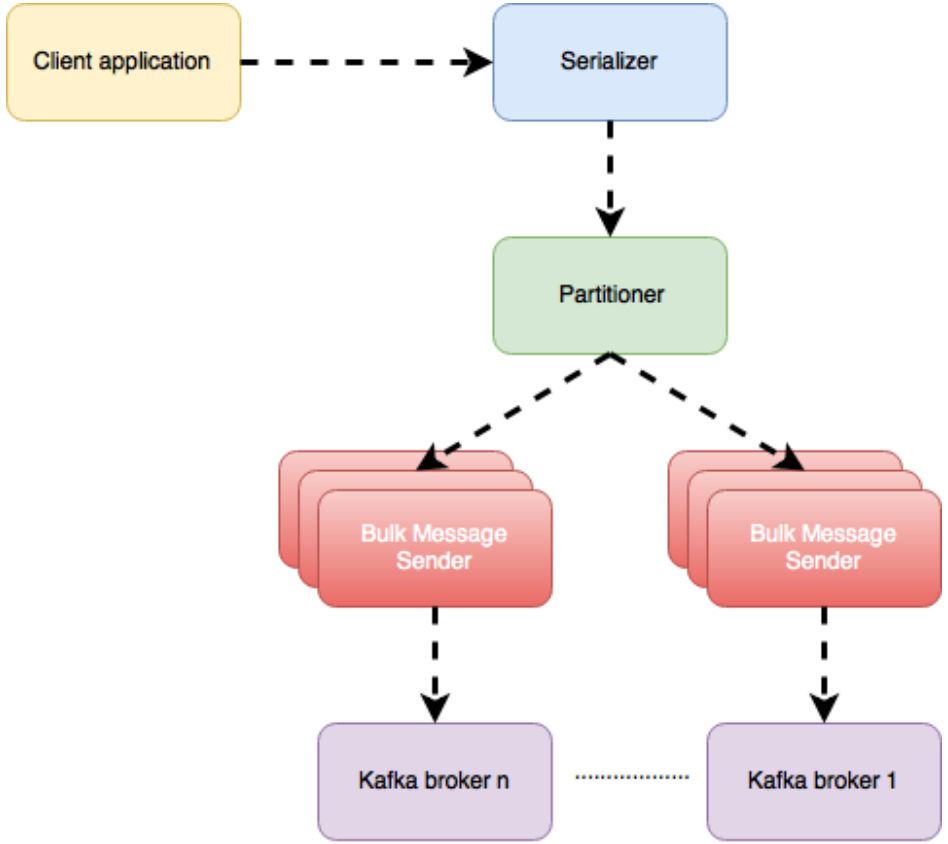
- Kafka Broker 集群受 Zookeeper 管理。所有的 Kafka Broker 節點一起去 Zookeeper 上註冊一個臨時節點，因為只有一個 Kafka Broker 會註冊成功，其他的都會失敗，所以這個成功在 Zookeeper 上註冊臨時節點的這個 Kafka Broker 會成為 Kafka Broker Controller，其他的 Kafka broker 叫 Kafka Broker follower。
- 當這個 Kafka broker controller 終止了，在 zookeeper 上面的那個臨時節點就會消失，此時所有的 Kafka broker 又會一起去 Zookeeper 上註冊一個臨時節點，因為只有一個 Kafka Broker 會註冊成功，其他的都會失敗，所以這個成功在 Zookeeper 上註冊臨時節點的這個 Kafka Broker 會成為 Kafka Broker Controller，其他的 Kafka broker 叫 Kafka Broker follower

Producer



Producer

- Producer 發送時進僅需要指定具體的 topic name 和 message 內容，即可將訊息發送到 topic 的某區
- Kafka 根據指定的 key 或自己的演算法，由依 Partition 機制 (kafka.producer.partitioner) 選擇將其存儲到哪一個 Partition



Producer

- key 為 null 時 Kafka 會將消息發送給哪個分區 ??
  - 如果 key 為 null，會從 sendPartitionPerTopicCache 查選緩存的分區 (依參數 "topic.metadata.refresh.ms" 的設定頻率刷新)。如果沒有，隨機選擇一個分區。

```
private def getPartition(topic: String, key: Any, topicPartitionList: Seq[PartitionAndLeader]): I
    val numPartitions = topicPartitionList.size
    if(numPartitions <= 0)
      throw new UnknownTopicOrPartitionException("Topic " + topic + " doesn't exist")
    val partition =
      if(key == null) {
        // If the key is null, we don't really need a partitioner
        // So we look up in the send partition cache for the topic to decide the target partitior
        val id = sendPartitionPerTopicCache.get(topic)
        id match {
          case Some(partitionId) =>
            // directly return the partitionId without checking availability of the leader,
            // since we want to postpone the failure until the send operation anyways
            partitionId
          case None =>
            val availablePartitions = topicPartitionList.filter(_.leaderBrokerIdOpt.isDefined)
            if (availablePartitions.isEmpty)
              throw new LeaderNotAvailableException("No leader for any partition in topic " + top
            val index = Utils.abs(Random.nextInt) % availablePartitions.size
            val partitionId = availablePartitions(index).partitionId
            sendPartitionPerTopicCache.put(topic, partitionId)
            partitionId
```

Producer

- Kafka 中有三種發送消息的方式：
  - 只發不管結果 (fire-and-forget)：只調用接口發送消息到 Kafka 服務器，但不管成功寫入與否。由於 Kafka 是高可用的，因此大部分情況下消息都會寫入，但在異常情況下會丟消息。
  - 同步發送 (Synchronous send)：調用 send() 方法返回一個 Future 對象，我們可以使用它的 get() 方法來判斷消息發送成功與否。
  - 異步發送 (Asynchronous send)：調用 send() 時提供一個回調方法，當接收到 broker 結果後回調此方法。

Fire and Forget


```
ProducerRecord<String, String> record =
    new ProducerRecord<>("CustomerCountry", "Precision Products", "France");

try {
    // 此方法返回 RecordMetadata, 但是這裡忽略了返回值。正式環境一般不適用此方式
    producer.send(record);

} catch (Exception e) {
    e.printStackTrace();
}
```

Synchronous


```
ProducerRecord<String, String> record =
    new ProducerRecord<>("CustomerCountry", "Precision Products", "France");

try {
    // 發送訊息並等待回應的 RecordMetadata
    producer.send(record).get();

} catch (Exception e) {
    e.printStackTrace();
}
```

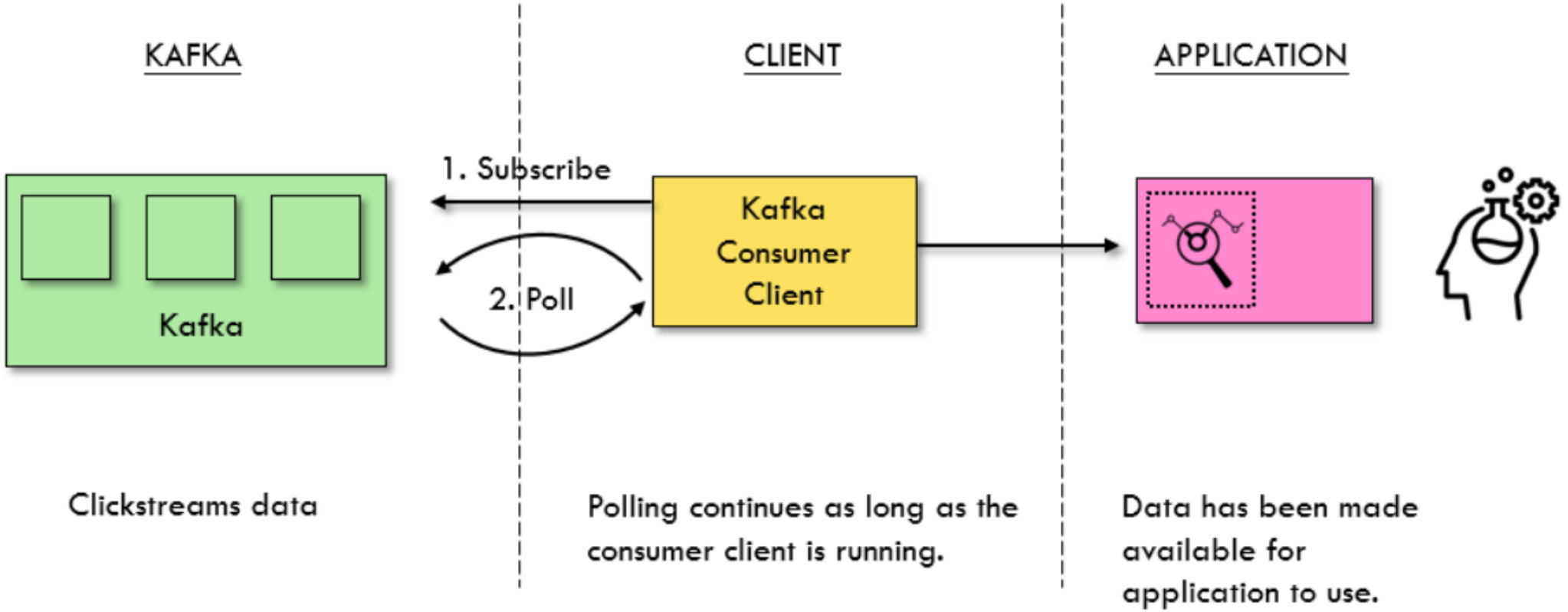
Asynchronous


```
private class DemoProducerCallback implements Callback {
    @Override
    public void onCompletion(RecordMetadata recordMetadata, Exception e) {
     if (e != null) {
         e.printStackTrace();
        }
    }
}

ProducerRecord<String, String> record =
    new ProducerRecord<>("CustomerCountry", "Biomedical Materials", "USA");

producer.send(record, new DemoProducerCallback());
```

Producer

- acks 參數控制在生產者認為寫入成功之前多少個 partition 副本必須接收到記錄。 可選值 all, -1, 0, 1， 默認值為 1
  - acks=0，生產者不會從 broker 得到任何回覆，所以不會知道消息是否丟失，生產者會盡快的發送消息，因此這種配置可以用來獲得非常高的吞吐量。
  - acks=1，當 leader 收到訊息時，生產者即會收到回應，如果消息不能寫入到 leader (leader 崩潰了，且新的 leader 還沒有選舉出來)，生產者將接收到一個錯誤響應，然後重新發送消息。
  - acks=all，等待數據完成副本的複製，等同於 -1。這是最安全的模式， 可以確保不只一個 broker 收到訊息，然而延遲也最高。

Consumer



Consumer Group

- consumer group 是 Kafka 提供的可擴展且具有容錯性的消費者機制。組內可以有多個消費者或消費者實例(consumer instance)，它們共享一個公共的ID，即 group ID。
- 組內的所有消費者協調在一起來消費訂閱主題(subscribed topics) 的所有分區 (partition)。每個分區只能由同一個消費組內的一個 consumer 來消費。

Consumer Group

- consumer group下可以有一個或多個 consumer instance, consumer instance 可以是一個 process，也可以是一個 thread
- group.id 是一個字符串，唯一標識一個 consumer group
- consumer group 下訂閱的 topic 下的每個分區只能分配給某個 group 下的一個 consumer (該分區還可以被分配給其他 group)

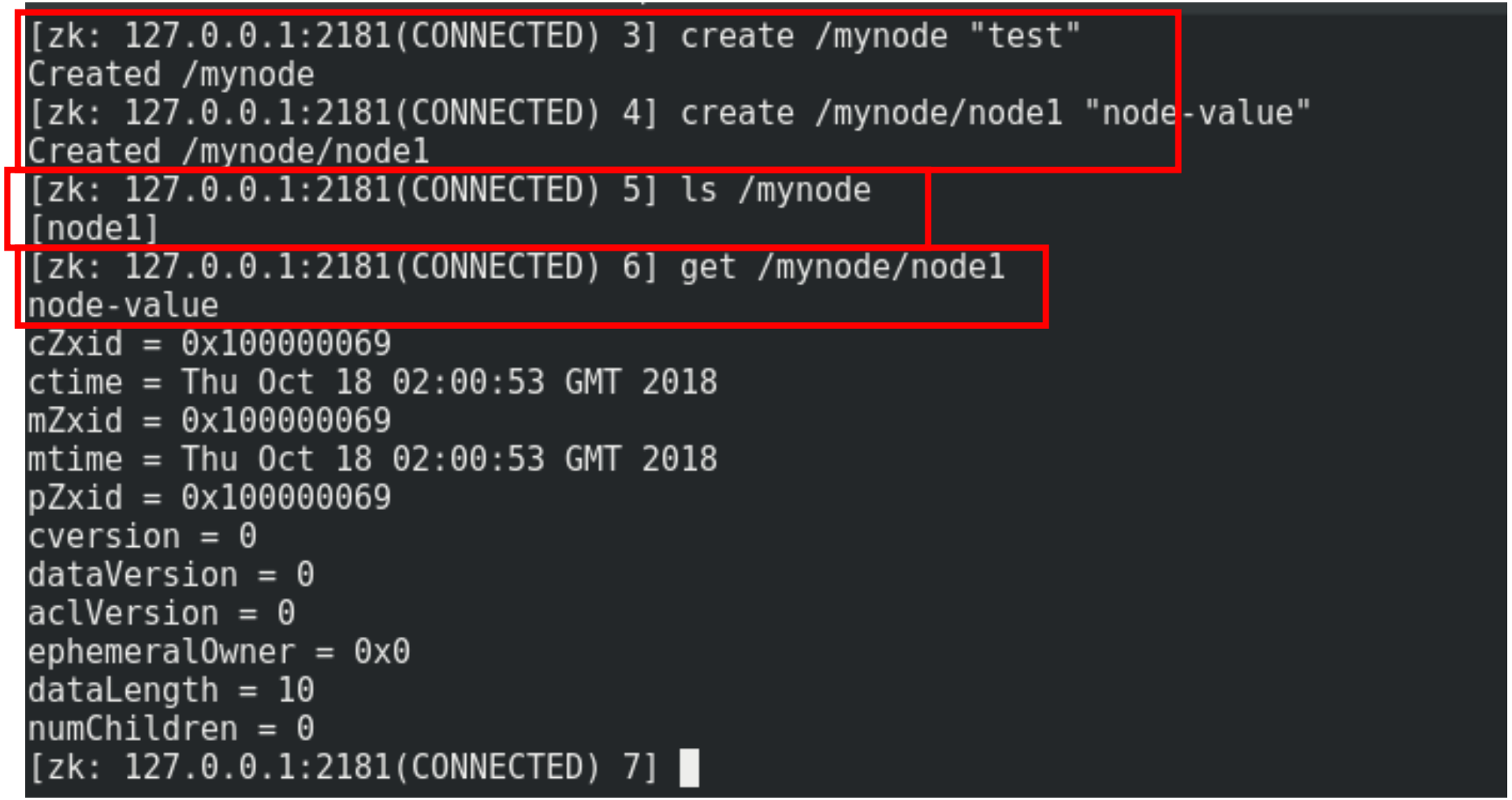
Zookeeper

- Apache ZooKeeper 是 Apache 軟體基金會的一個軟體專案，他為大型分散式計算提供開源的分散式組態設定服務、同步服務和命名註冊。ZooKeeper 曾經是 Hadoop 的一個子專案，但現在是一個獨立的頂級專案

Zookeeper

- Kafka 使用 Zookeeper 來保存 broker、topic 與 partition 的 metadata。對於一個包含多個節點的 zookeeper 群組來說，kafka 的集群流量並不算多，僅用於構造 consumer group 或 cluster 本身。
- 在 kafka v0.9 之前，除 broker 外，consumer 也會使用 zookeeper 來保存訊息，如各 partition 的 offset

Zookeeper



Offset

- Consumer 在消費的過程中需要記錄自己消費了多少數據，即消費位置信息。在 Kafka 中這個位置信息有個專門的術語：位移(offset)。
- Kafka 中每個 consumer group 保存自己的位移信息

Offset Commit

- 早期版本的位移資訊是提交到 zookeeper 中的，但 zookeeper 並不適合進行大批量的讀寫操作。因此 kafka 提供了另一種解決方案：增加 __consumer_offsets topic，將 offset 信息寫入這個 topic。

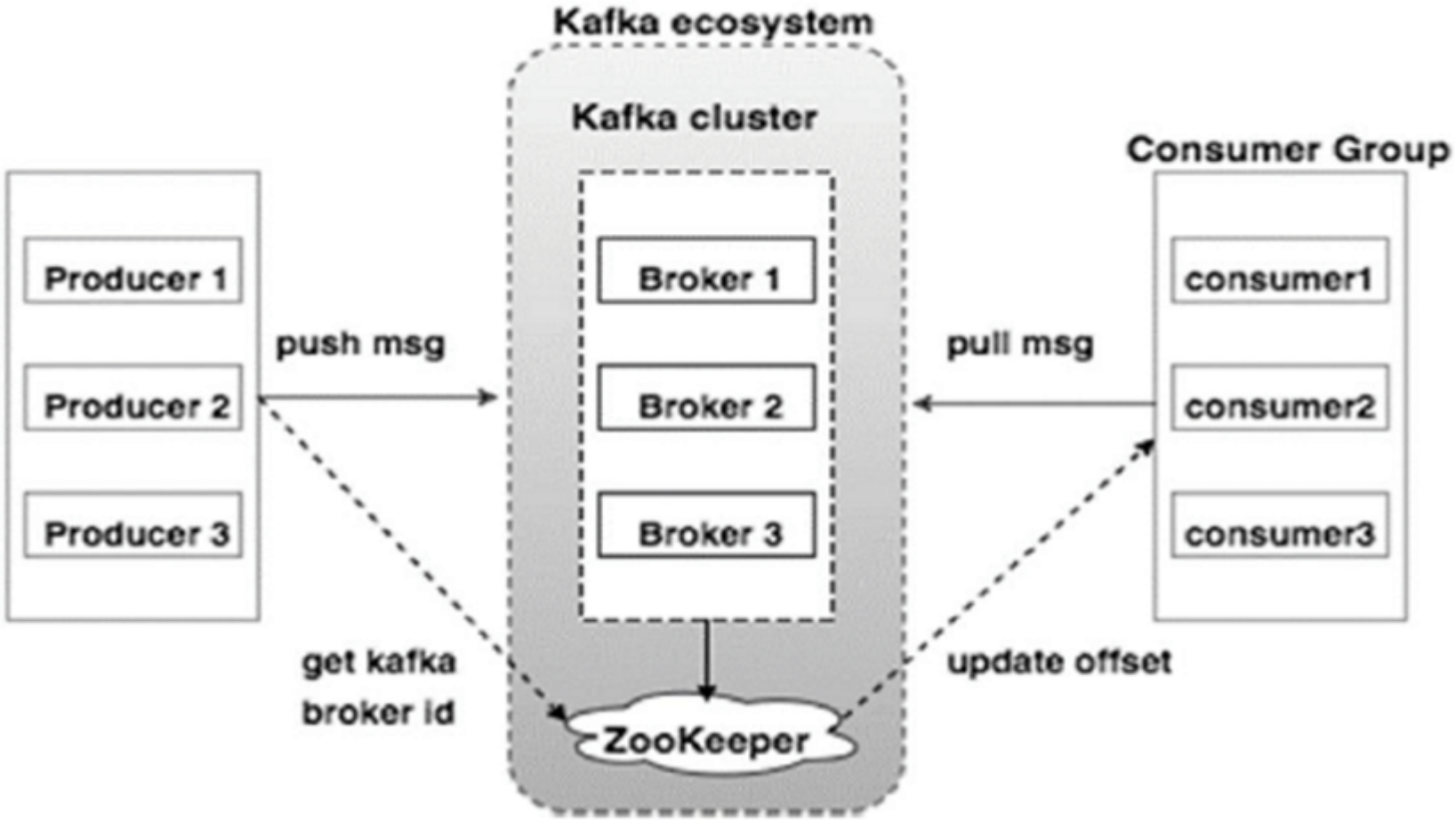
Consumer Group Rebalance

- Rebalance 本質上是一種協議，規定了一個 consumer group下的所有 consumer 如何達成一致來分配訂閱 topic 的每個分區。
- 比如某個 group 下有 20 個 consumer，它訂閱了一個具有 100 個 partitions 的 topic。正常情況下，Kafka 平均會為每個 consumer 分配 5 個 partition。這個分配的過程就叫 rebalance。

Rebalance Happening Time

- 組成員發生變更，新 consumer 加入組、已有 consumer 主動離開組或已有 consumer 崩潰了
- 訂閱主題數發生變更 (if use regular expression to subscribe topics)
- 訂閱主題的分區數 (partition numbers) 發生變更

Kafka Cluster Architecture

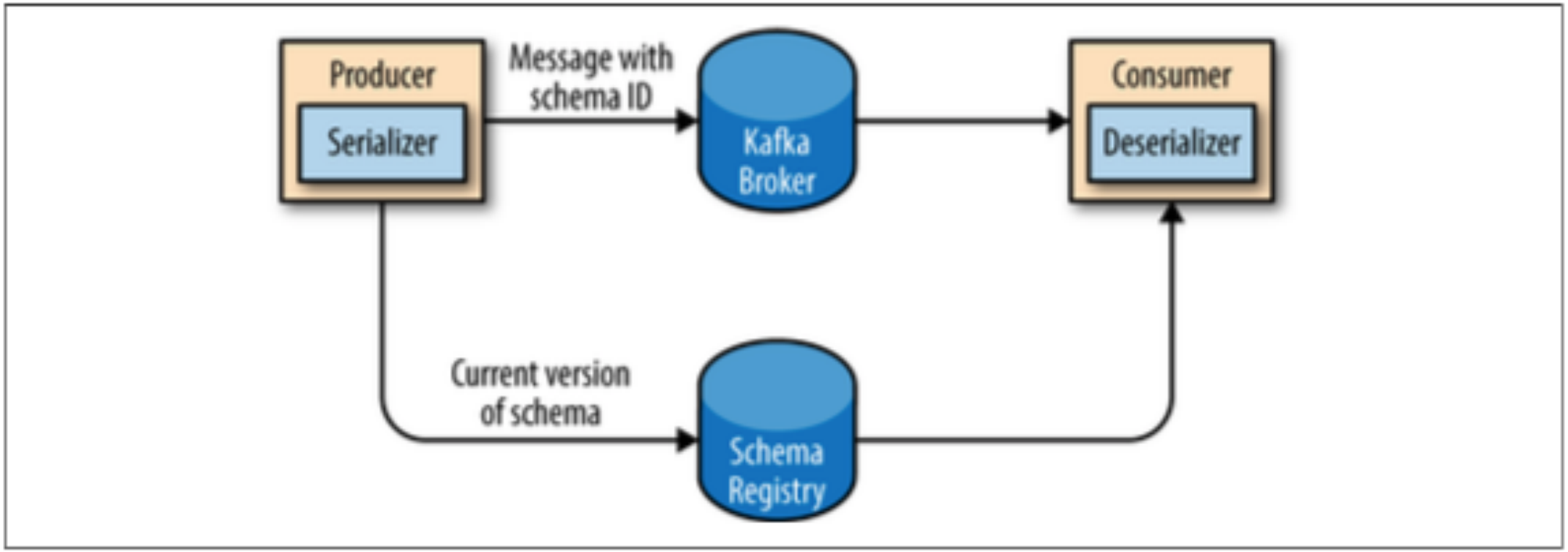


Kafka Ecosystems

Schema Registry

- 當使用 Avro 序列化成文件時，我們可以將數據的結構添加到文件中；但對於 Kafka，如果對於每條 Avro 消息附上消息結構，將增加差不多一倍的開銷。
- 使用模式註冊中心（Schema Registry）的架構模式，將消息的結構存儲在註冊中心，這樣消費者可以從註冊中心獲取數據的結構

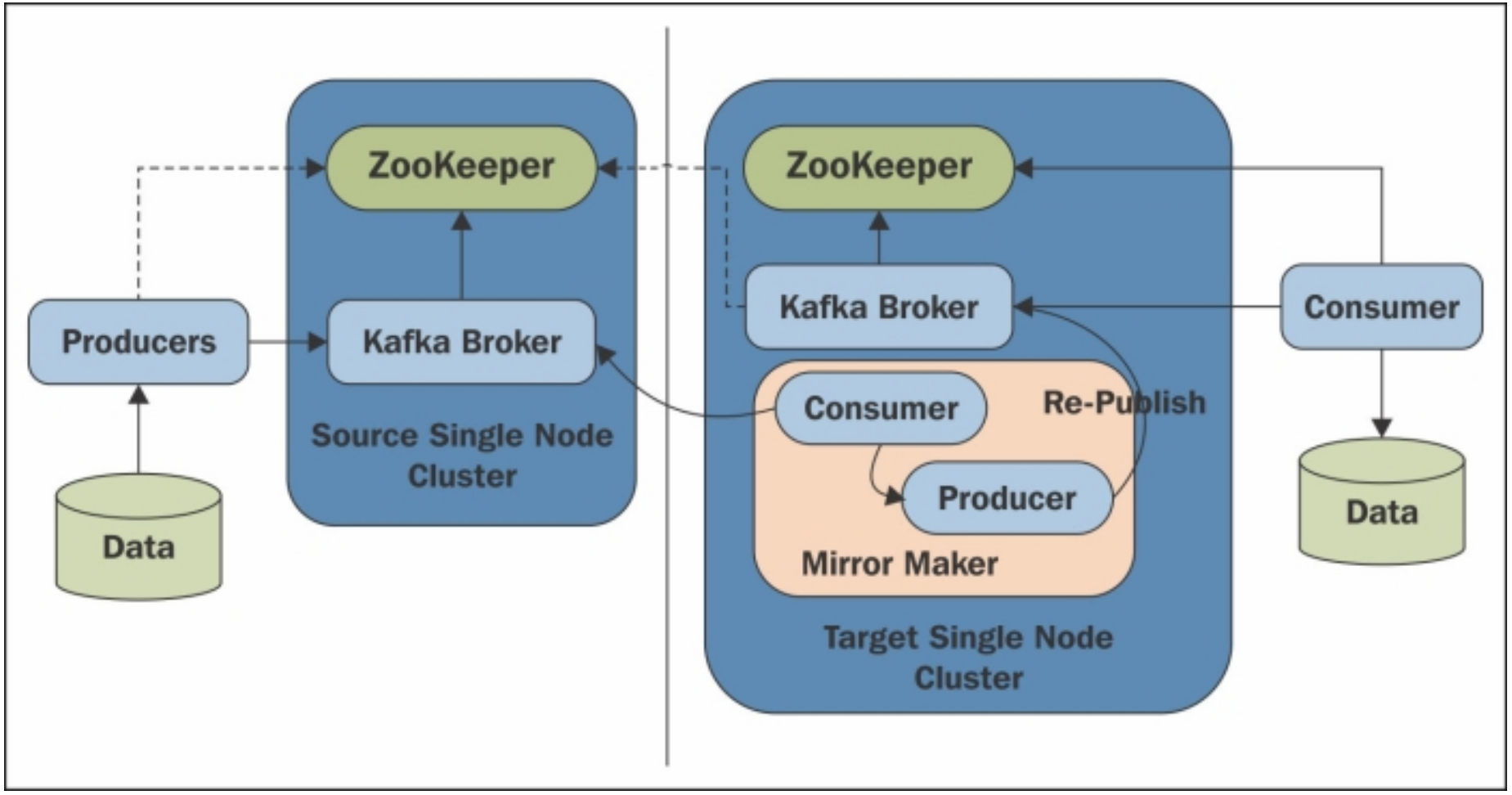
Schema Registry



Mirror Maker

- 用於在兩個數據中心之間的鏡像作業
- 概括來說 Mirror Maker 就是 kafka 生產者與消費者的一個整合，通過 consumer 從源 Kafka 集群消費數據，然後通過 producer 將數據重新推送到目標 Kafka 集群

Mirror Maker



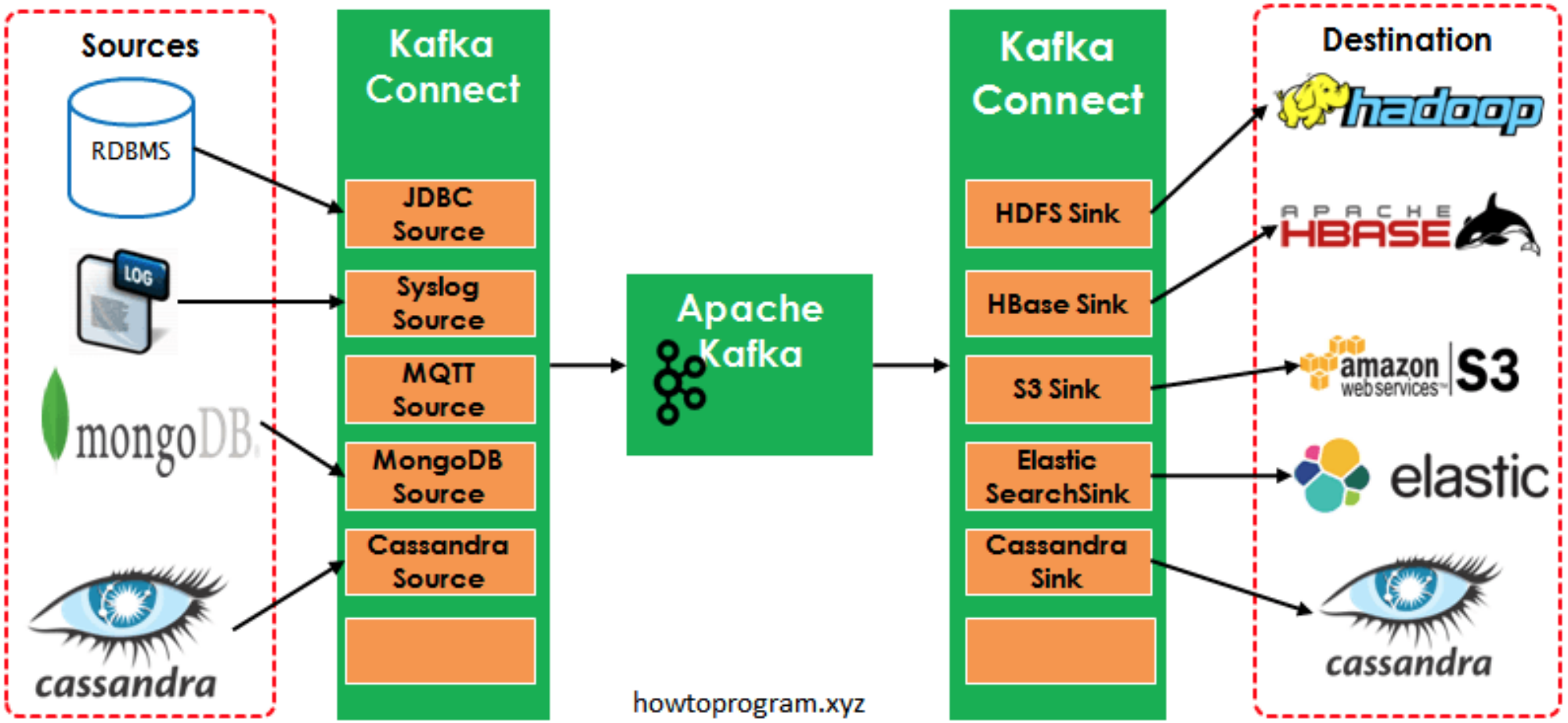
Mirror Maker Applications

- 機房搬遷，在完成機房搬遷移前，老機房數據寫入完成後再同步至新機房 kafka 集群
- 彙整不同國家的各分公司數據資料庫，集中彙整獲得整體的資料中心

Kafka Connect

- Kafka 0.9+ 增加了一個新的特性 Kafka Connect，可以更方便的創建和管理數據流管道。它為 Kafka 和其它系統創建規模可擴展的、可信賴的流數據提供了一個簡單的模型，通過 connectors 可以將大數據從其它系統導入到 Kafka 中，也可以從 Kafka 中導出到其它系統

Kafka Connect



Q & A

– THE PATHFINDER TO OPENSOURCE –

info@athemaster.com
FB: Athemaster