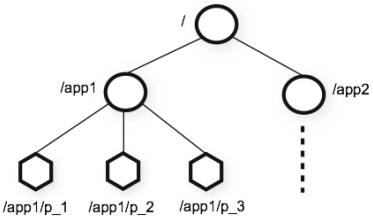
Ordering Service

Apache Zookeeper

- 분산 시스템에서는 분산된 시스템간의 정보를 어떻게 공유해야 하며 클러스터에 있는 서버들의 상태를 체크할 필요가 있다. 또한, 분산된 서버들 간에 동기화를 위한 Lock 을 처리해야 한다. 이 역할을 하는 시스템을 Coordination service 라 한다.

- 아파치 재단의 공개 소프트웨어 프로젝트
- Coordination service 의 일종
    - Coordination service : 분산 시스템을 코디네이션 하는 용도로 디자인 되었다.
        - 자체적으로 장애에 대한 대응 가능 → 데이터 유실 없이 fail over/fail back 가능
        - 빠른 Data Access → 자체적으로 Clustering 제공

- 아키텍처
    - Coordination service 는 분산 시스템 내에서 중요한 상태 정보나 설정 정보 등을 유지하기 때문에, Coordination service 의 장애는 전체 시스템의 장애를 유발한다. 따라서 이중화 등을 통하여 고 가용성을 제공해야 한다. -> Zookeeper 는 이러한 특성을 잘 제공
    - Zookeeper 는 디렉토리 구조기반으로 Znode(Zookeeper node)라는 데이터 저장 객체(key-value) 제공 → 해당 객체에 데이터의 입출력 기능만 제공

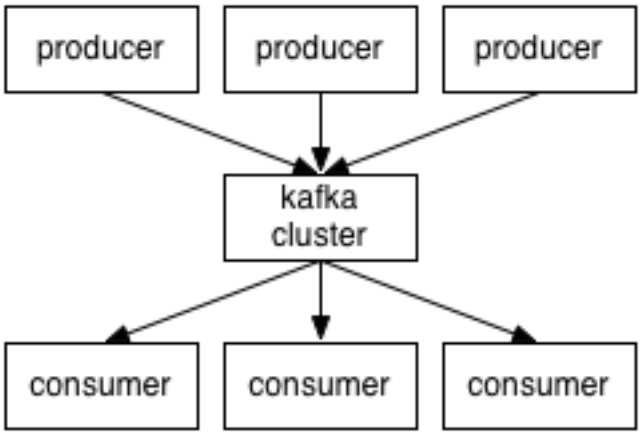
- 데이터 모델
    - 

디렉토리 구조의 각 노드에 데이터를 저장
    - Znode types
        - Persistent Node : 노드에 저장된 데이터는 일부러 삭제하지 않는 한 영구적
        - Ephemeral Node : 노드를 생성한 클라이언트의 세션이 연결되어 있어야만 유효
        - Sequencd Node : 노드를 생성할 때 자동으로 sequence 번호가 붙는 노드

Apache Kafka

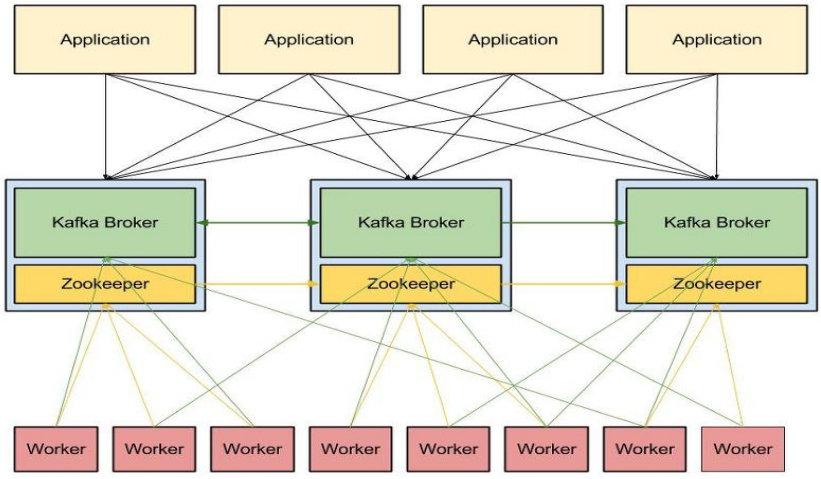
- 개요
  - LinkedIn 에서 개발된 분산 메시징 시스템
  - 대용량의 실시간 처리에 특화된 아키텍처 설계를 통하여 우수한 TPS 를 보여준다.

- 기본 구성 요소
  - Publish-subscribe 모델을 기반으로 동작
  - Producer / Consumer / Broker 로 구성된다.



- 동작
  - Broker : topic 을 기준으로 메시지 관리 (분류)
  - Producer : 특정 topic 의 메시지를 생성한 뒤 해당 메시지를 Broker 에 전달한다.
  - Consumer : 구독하는 topic 의 메시지를 가져가서 처리한다.

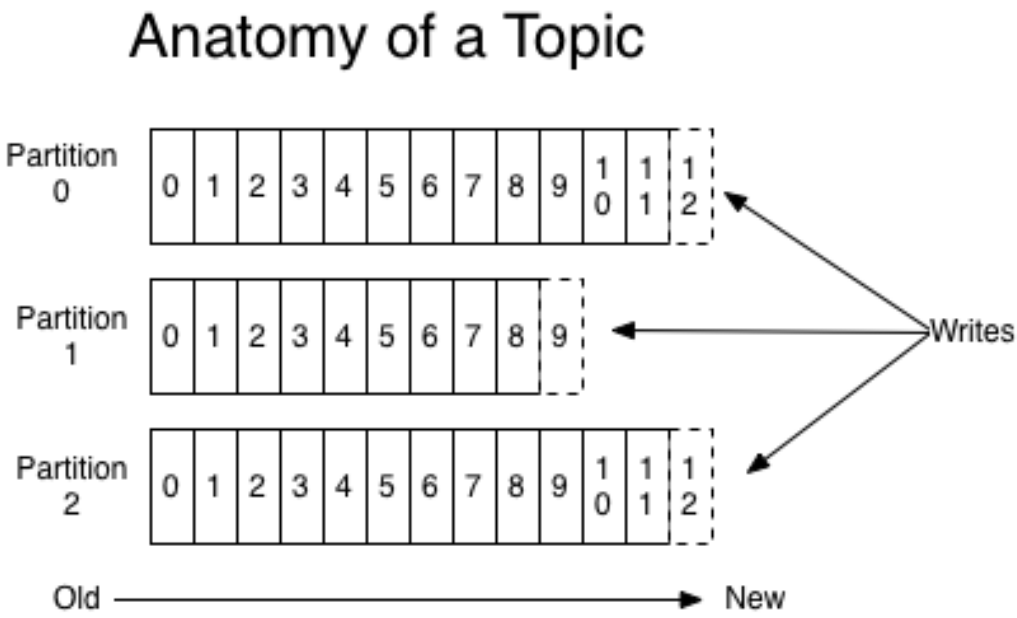
- 설계
  - Scale-out / High Availability 을 위하여 Broker 들이 클러스터로 구성되어 동작하도록 설계
    - Broker 가 1 개 밖에 없을 경우에도 클러스터로써 동작
  - 클러스터 내의 Broker 에 대한 분산 처리는 Apache Zookeeper 가 담당한다.



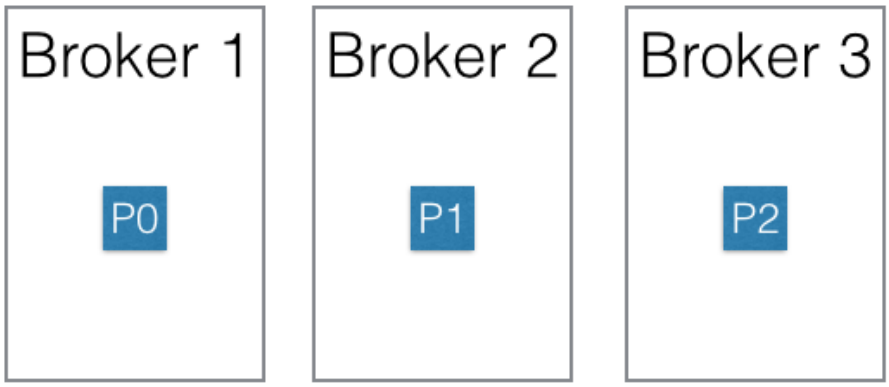
- 차별점 (기존 메시징 시스템 : ActiveMQ, RabbitMQ)
  - 대용량의 실시간 로그 처리에 특화되어 설계된 메시징 시스템으로써 기존 범용 메시징 시스템대비 TPS 가 매우 우수하다. 단, 특화된 시스템이기 때문에 범용 메시징 시스템에서 제공하는 다양한 기능들은 제공되지 않는다.

- 분산 시스템을 기본으로 설계 ⇒ 기존 메시징 시스템에 비해 분산 및 복제 구성이 쉽다.
- AMQP 프로토콜이나 JMS API 를 사용하지 않고 단순한 메시지 헤더를 지닌 TCP 기반의 프로토콜을 사용하여 프로토콜에 의한 오버헤드 감소
- Producer 가 Broker 에게 다수의 메시지를 전송할 때 각 메시지를 개별적으로 전송해야하는 기존 메시징 시스템과는 달리, 다수의 메시지를 Batch 형태로 Broker 에게 한 번에 전달할 수 있어 TCP/IP 라운드 트립 횟수를 줄일 수 있다.
- 메시지를 기본적으로 메모리에 저장하는 기존 메시징 시스템과는 달리 메시지를 파일 시스템에 저장한다.
    - 데이터의 durability 보장
    - 처리되지 않은 메시지가 쌓이면 성능이 크게 감소하는 기존 메시징 시스템과는 달리 Kafka 는 성능이 크게 감소하지 않는다.
    - 파일 시스템에 저장되기 때문에 실시간 처리뿐만 아니라 주기적인 Batch 작업에 사용할 데이터를 쌓아두는 용도로도 사용 가능하다.
    - Consumer 에 의해 처리된 메시지를 곧바로 삭제하는 기존 메시징 시스템과는 달리 처리된 메시지를 삭제하지 않고 파일 시스템에 그대로 두었다가 설정된 수명이 지나면 삭제한다.
      ⇒ 메시지 처리 도중 문제가 발생하였거나 처리 로직이 변경되었을 경우 Consumer 가 메시지를 처음부터 다시 처리(rewind)하도록 할 수 있다.
- 기존의 메시징 시스템에서는 Brokerr 가 Consumer 에게 메시지를 Push 해 주는 방식인데 반해, Kafka 는 consumer 가 Broker 로 부터 직접 메시지를 가지고 가는 Pull 방식으로 동작한다. ⇒ Consumer 는 자신의 처리능력만큼의 메시지만 Broker 로 부터 가져오기 때문에 최적의 성능을 낼 수 있다.
    - 기존의 Push 방식의 메시징 시스템에서는 Broker 가 직접 각 consumer 가 어떤 메시지를 처리해야 하는지 계산하고 어떤 메시지를 처리 중인지 트랙킹하였는데, Kafka 에서는 consumer 가 직접 필요한 메시지를 Broker 로부터 Pull 하므로 Broker 의 consumer / 메시지 관리에 대한 부담이 줄었다.
    - 메시지를 Pull 방식으로 가져오므로, 메시지를 쌓아두었다가 주기적으로 처리하는 batch consumer 의 구현이 가능하다.

- Topic & Partition [Broker]
  - Topic 은 Partition 이라는 단위로 쪼개어져 클러스터의 각 서버들에 분산되어 저장되고, 고가용성을 위하여 복제 (Replication) 설정을 할 경우 이 또한 Partition 단위로 각 서버들에 분산되어 복제되고 장애가 발생하면 Partition 단위로 fail over 가 수행된다.
  - 하나의 Topic 이 3 개의 Partition 에 분산되어 순차적으로 저장되는 모습

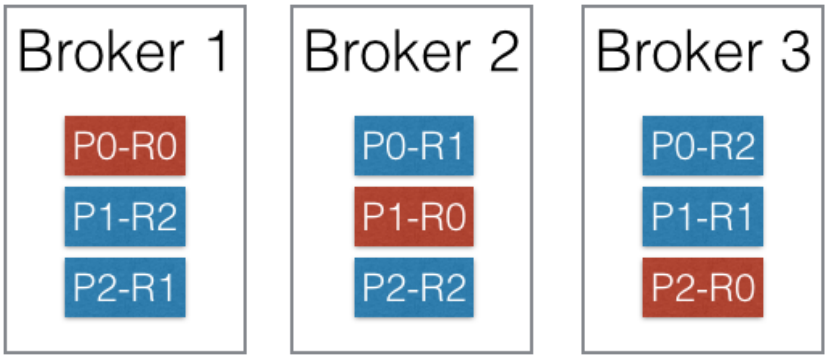


    -
    - 각 Partition 은 0 부터 1 씩 증가하는 offset 값을 메시지에 부여하는데 이 값은 각 Partition 내에서 메시지를 식별하는 ID 로 사용된다. Offset 값은 Partition 마다 별도로 관리되므로 Topic 내에서 메시지를 식별할 때는 Partition 번호와 Offset 값을 함께 사용한다.
  - Partition 의 분산



    - 3 개의 Broker 로 이루어진 클러스터에서 하나의 topic 이 3 개의 Partition P0, P1, P2 로 분산되어 저장되어 있다.
    - Producer 가 메시지를 실제로 어떤 Partition 으로 전송할지는 사용자가 구현한 Partition 분배 알고리즘에 의해 결정된다.
      - Ex) 라운드-로빈 방식의 Partition 분배 알고리즘을 구현할 경우 각 Partition 에 메시지를 균등하게 분배하도록 하거나, 메시지의 키를 활용하여 알파벳 A 로 시작하는 키를 가진 메시지는 P0 에만 전송하고, B 로 시작하는 키를 가진 메시지는 P1 에만 전송하는 형태의 구성도 가능하다.

- Partition 의 복제
  - 고가용성을 위하여 각 Partition 을 복제하여 클러스터에 분산시킬 수 있다.



[Topic 의 Replication factor 를 3 으로 설정한 상태의 클러스터]
[각 Partition 들은 3 개의 replica 를 가지며 각 replica 는 R0,R1,R2 로 표시된다]

  - Replication factor 를 N 으로 설정할 경우 N 개의 replica 는 1 개의 leader 와 N-1 개의 follower 로 구성된다. 위의 그림에서는 각 Partition 마다 하나의 Leader(붉은색)가 존재하며 2 개의 follower(푸른색)가 존재한다.

  - 각 Partition 에 대한 읽기와 쓰기는 모두 leader 에서 이루어지며, follower 는 단순히 leader 를 복제하기만 한다. 만약 leader 에 장애가 발생할 경우 follower 중 하나가 새로운 leader 가 된다. Kafka 의 복제 모델인 ISR 모델은 f+1 개의 replica 를 가진 topic 이 f 개의 장애까지 버틸 수 있다.

  - Leader 에서만 읽기와 쓰기를 수행한다고 하면 부하 분산이 되지 않는다고 생각할 수 있는데, 각 partition 의 leader 가 클러스터 내의 broker 들에 균등하게 분배되도록 알고리즘이 설계되어 있기 때문에 부하는 자연스럽게 분산된다.

- Consumer 와 Consumer Group
  - 메시징 모델은 크게 Queue 모델과 Publish-subscribe 모델로 나뉜다
    - 큐 모델 : 메시지가 쌓여있는 큐로부터 메시지를 가져와서 consumer pool 에 있는 consumer 중 하나에 메시지를 할당하는 방식
    - Publish-Subscribe 모델 : topic 을 구독하는 모든 consumer 에게 메시지를 브로드캐스팅하는 방식
  - Kafka 에서는 consumer group 이라는 개념을 도입하여 두가지 모델을 publish-subscribe 모델로 일반화하였다. Partition 은 consumer group 당 오로지 하나의 consumer 의 접근만을 허용하며, 해당 consumer 를 partition owner 라고 부른다. 따라서 동일한 consumer group 에 속하는 consumer 끼리는 동일한 Partition 에 접근할 수 없다.
  - Consumer group 을 구성하는 consumer 의 수가 Partition 의 수보다 작으면 하나의 Consumer 가 여러 개의 partition 을 소유하게 되고, 반대의 경우 여분의 consumer 는 메시지를 처리하지 않게 되므로 partition 개수와 consumer 수의 적절한 설정이 필요하다.