Kafka介绍

概述

- Kafka是Linkedin于2010年12月份开源的消息系统，它主要用于处理活跃的流式数据。活跃的流式数据在web网站应用中非常常见，这些数据包括网站的pv、用户访问了什么内容，搜索了什么内容等。这些数据通常以日志的形式记录下来，然后每隔一段时间进行一次统计处理。传统的日志分析系统提供了一种离线处理日志信息的可扩展方案，但若要进行实时处理，通常会有较大延迟。而现有的消（队列）系统能够很好的处理实时或者近似实时的应用，但未处理的数据通常不会写到磁盘上，这对于Hadoop之类（一小时或者一天只处理一部分数据）的离线应用而言，可能存在问题。Kafka正是为了解决以上问题而设计的，它能够很好地离线和在线应用。

应用场景

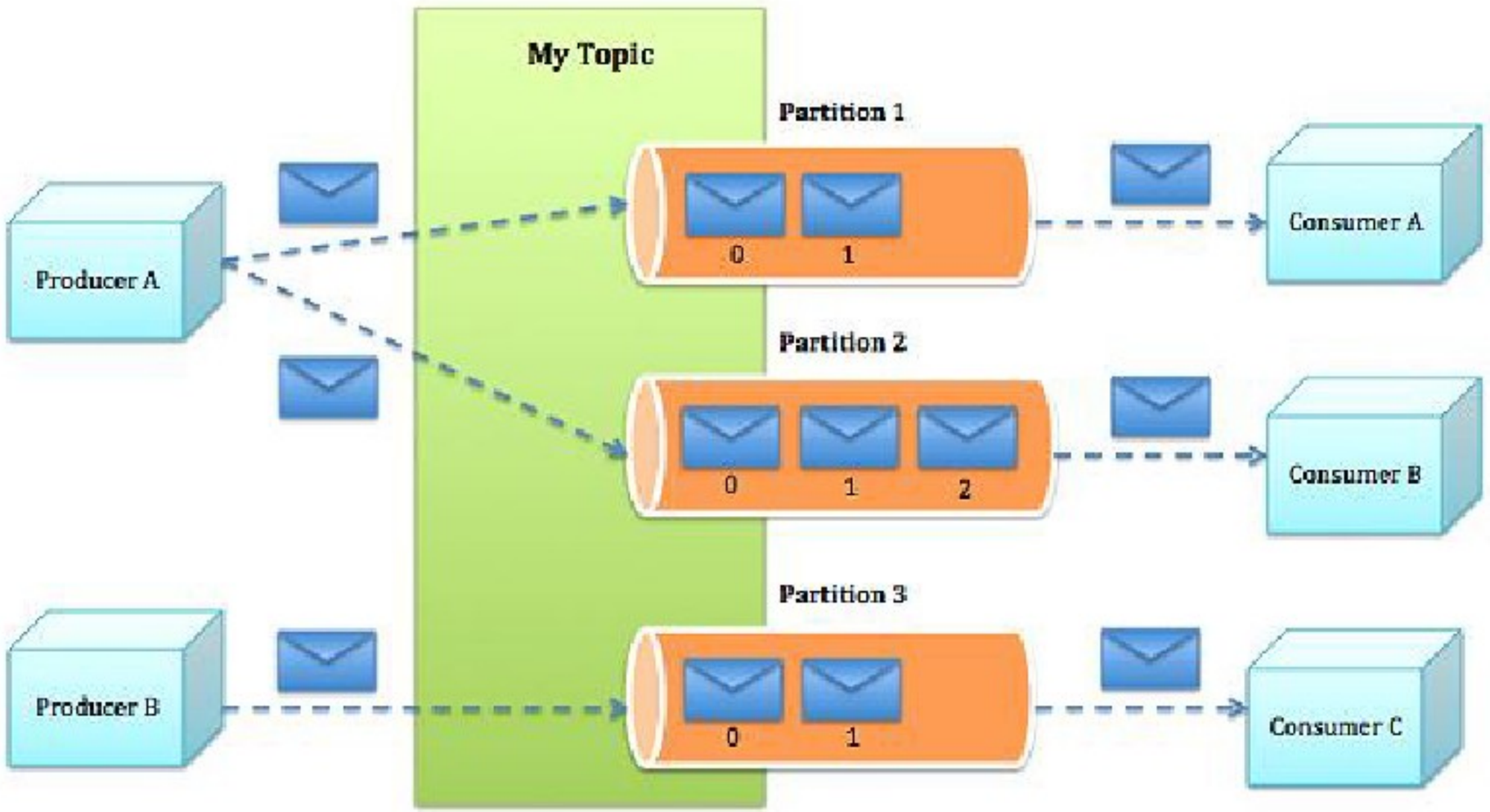
- 消息投递：能够很好的代替传统的message broker。它提供了更强大的吞吐量，内建分区，复本，容错等机制来解决大规模消息处理型应用程序。
- 用户活动追踪：通过按类型将每个web动作发送到指定topic,然后由消费者去订阅各种topic,处理器包括实时处理，实时监控，加载到hadoop或其他离线存储系统用于离线处理等。
- 日志聚合：把物理上分布在各个机器上的离散日志数据聚集到指定区域（如HDFS或文件服务器等）用来处理。

设计特点

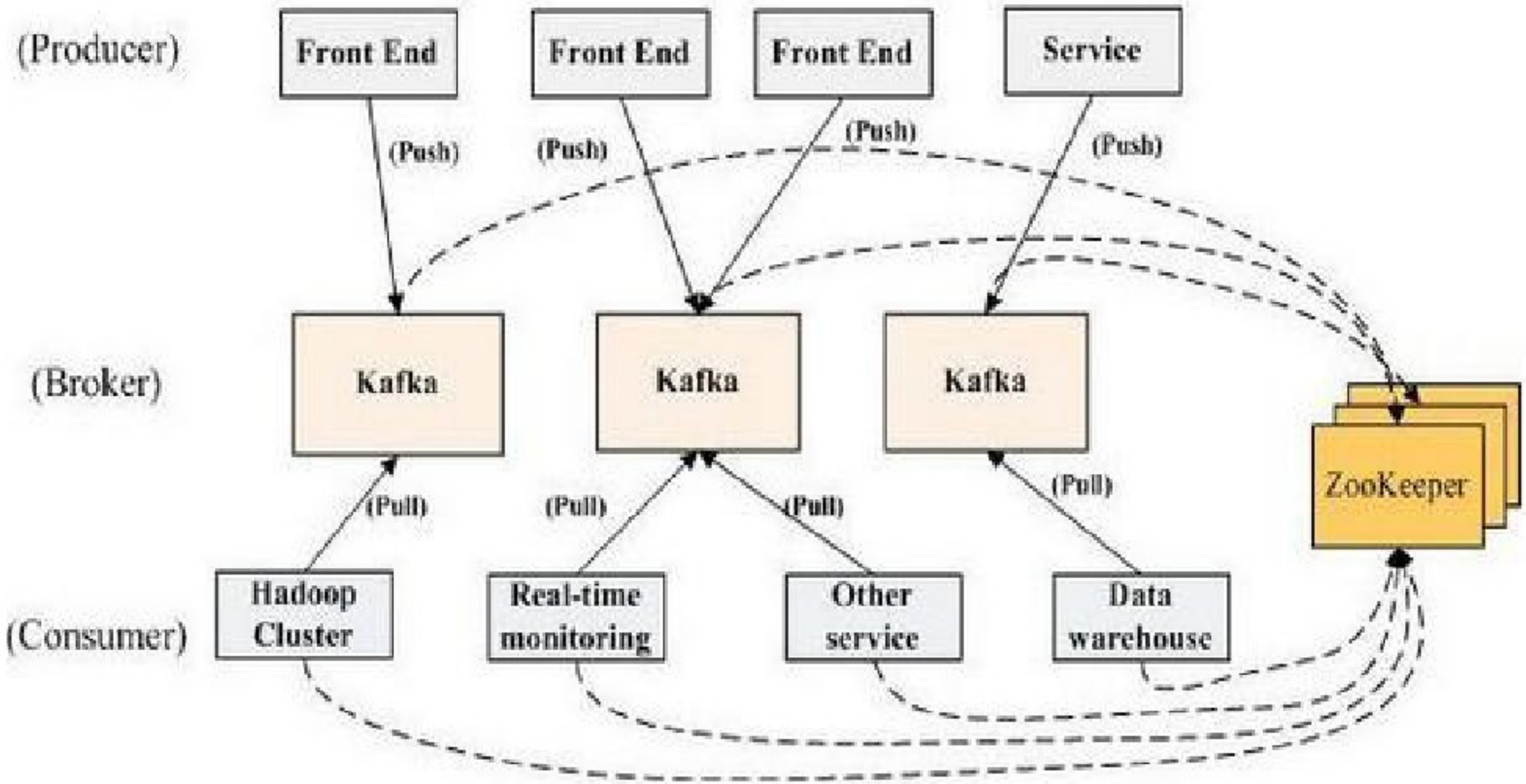
- 消息持久化：通过O(1)的磁盘数据结构提供消息的持久化。
- 高吞吐量：每秒百万级的消息读写。
- 分布式： Kafka是显式分布式的，多个producer、consumer和broker可以运行在一个大的集群上，作为一个逻辑整体对外提供服务。对于consumer，多个consumer可以组成一个group，这个message只能传输给某个group中的某一个consumer。
- 多客户端支持：java、php、ruby、pathon、c、c++ 。
- 实时性：生产者生产的message立即被消费者可见。

kafka组件

Kafka内是分布式的，一个Kafka集群通常包括多个broker。为了均衡负载，将话题分成多个分区，每个broker存储一或多个分区。多个生产者和消费者能够同时生产和获取消息。

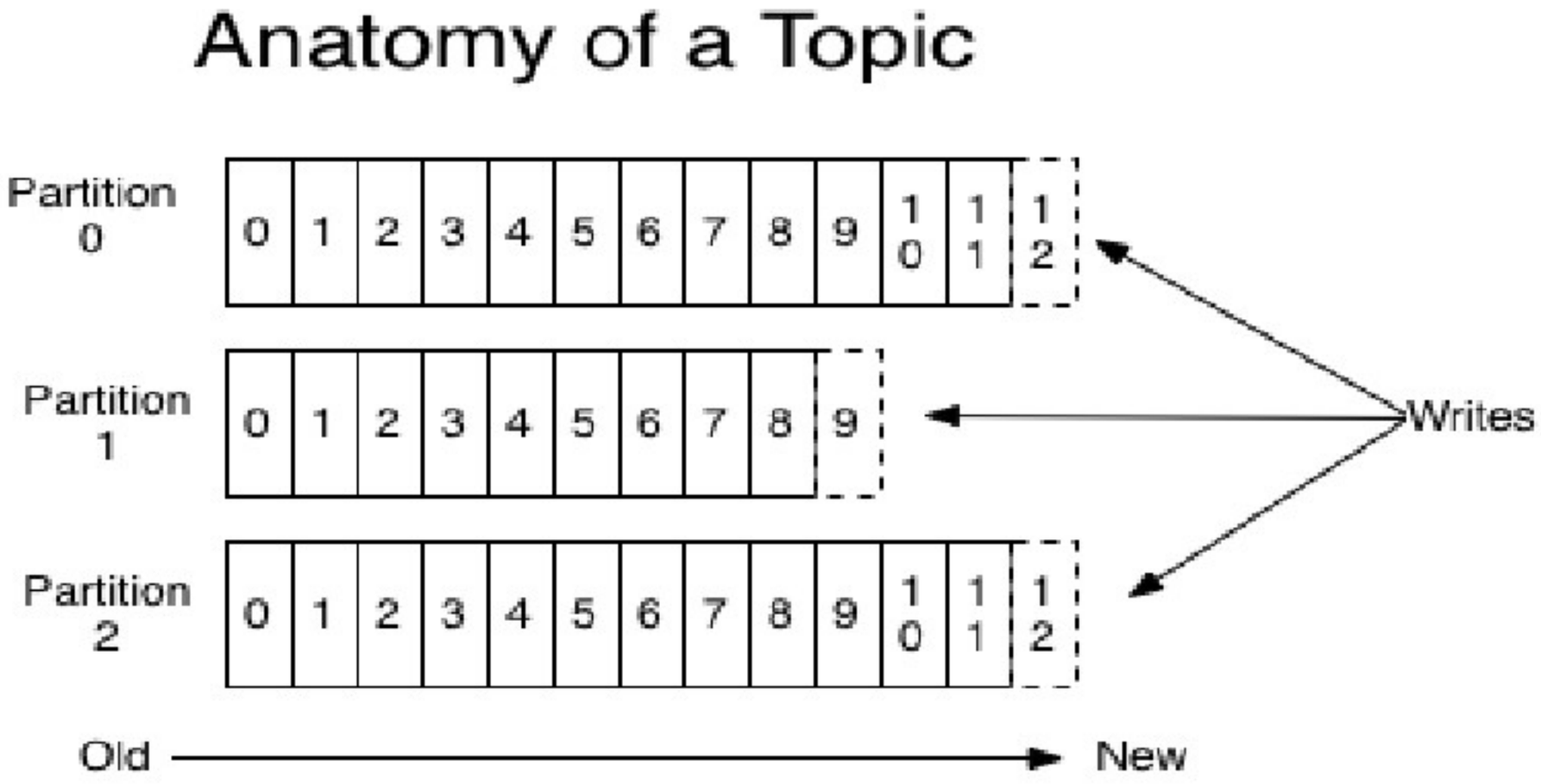


kafka组件


(Producer)
Front End
Front End
Front End
Service
(Push)
(Push)
(Push)
(Push)
(Broker)
Kafka
Kafka
Kafka
ZooKeeper
(Pull)
(Pull)
(Pull)
(Pull)
(Consumer)
Hadoop Cluster
Real-time monitoring
Other service
Data warehouse

topics

- 一个topic是一个用于发布消息的分类或feed名，kafka集群使用分区的日志，每个分区都是有顺序且不变的消息序列。commit的log可以不断追加。消息在每个分区中都分配了一个叫offset的id序列来唯一识别分区中的消息。



topics

- 无论发布的消息是否被消费，kafka都会持久化一定时间（可配置）。

- 在每个消费者都持久化这个offset在日志中。通常消费者读消息时会使offset值线性的增长，但实际上其位置是由消费者控制，它可以按任意顺序来消费消息。比如复位到老的offset来重新处理。

- 每个分区代表一个并行单元。

message

- message（消息）是通信的基本单位，每个producer可以向一个topic（主题）发布一些消息。如果consumer订阅了这个主题，那么新发布的消息就会广播给这些consumer。

- message format：
  message length : 4 bytes (value: 1+4+n)
  "magic" value : 1 byte
  crc : 4 bytes
  payload : n bytes

producer

- 生产者可以发布数据到它指定的topic中，并可以指定在topic里哪些消息分配到哪些分区（比如简单的轮流分发各个分区或通过指定分区语义分配key到对应分区）

- 生产者直接把消息发送给对应分区的broker，而不需要任何路由层。

- 批处理发送，当message积累到一定数量或等待一定时间后进行发送。

consumer

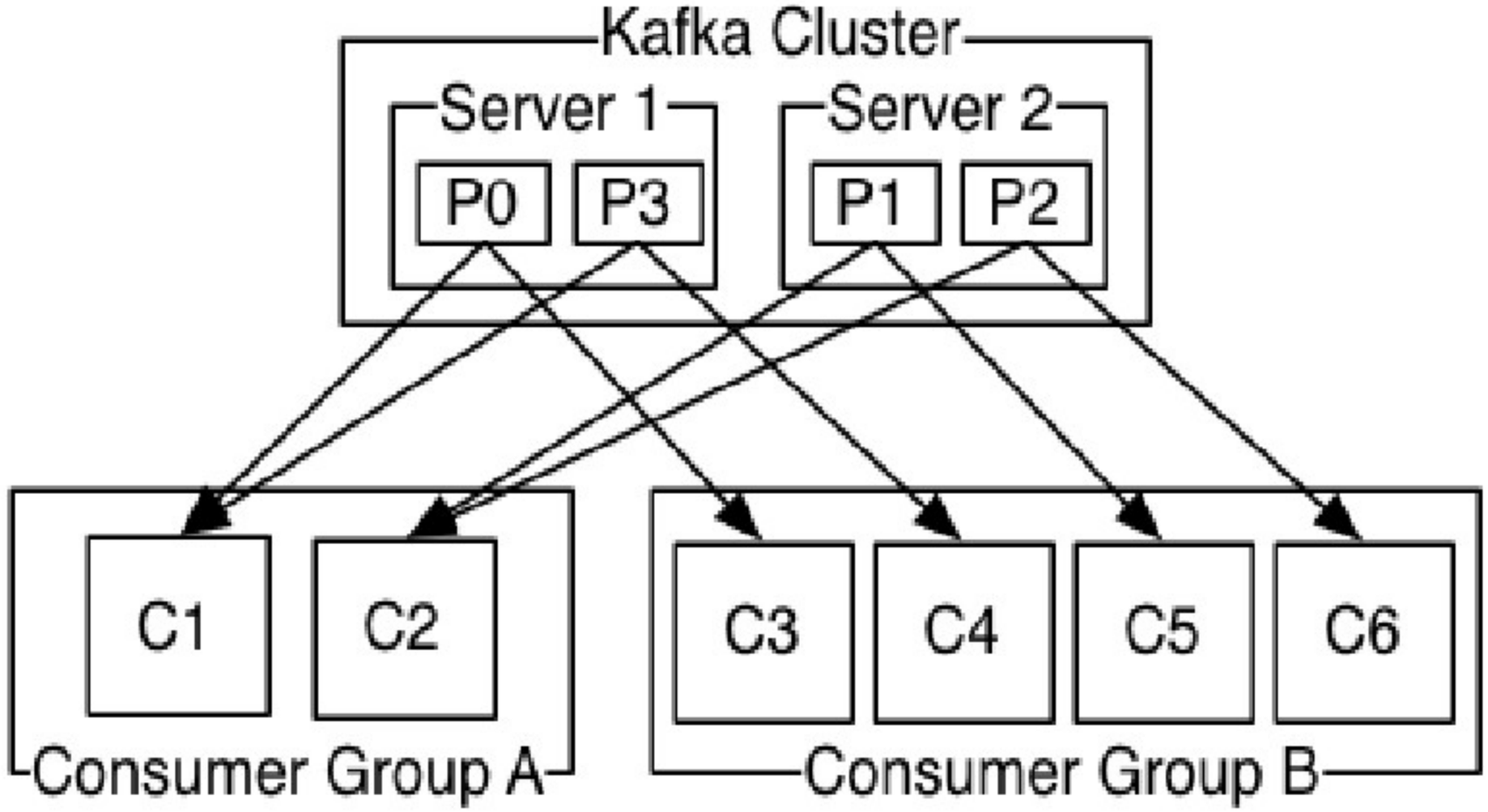
- 传统消费一般是通过queue方式（消息依次被感兴趣的消费者接受）和发布订阅的方式（消息被广播到所有感兴趣的消费者）。kafka采用一种更抽象的方式：消费组（consumer group）来囊括传统的两种方式。首先消费者标记自己一个消费组名。消息将投递到每个消费组中的某一个消费者实例上。如果所有的消费者实例都有相同的消费组，这样就像传统的queue方式。如果所有的消费者实例都有不同的消费组，这样就像传统的发布订阅方式。消费组就好比是个逻辑的订阅者，每个订阅者由许多消费者实例构成(用于扩展或容错)。
- 相对于传统的消息系统，kafka拥有更强壮的顺序保证。kafka由于topic采用了分区，故能够很好在多个消费者进程操作时保证顺序性和负载均衡。

consumer

- 传统消费一般是通过**queue**方式（消息依次被感兴趣的消费者接受）和发布订阅的方式（消息被广播到所有感兴趣的消费者）。**kafka**采用一种更抽象的方式：消费组（**consumer group**）来囊括传统的两种方式。首先消费者标记自己一个消费组名。消息将投递到每个消费组中的某一个消费者实例上。如果所有的消费者实例都有相同的消费组，这样就像传统的**queue**方式。如果所有的消费者实例都有不同的消费组，这样就像传统的发布订阅方式。消费组就好比是个逻辑的订阅者，每个订阅者由许多消费者实例构成(用于扩展或容错)。

- 相对于传统的消息系统，**kafka**拥有更强壮的顺序保证。**kafka**由于**topic**采用了分区，故能够很好在多个消费者进程操作时保证顺序性和负载均衡。如下图：

consumer


Kafka Cluster
Server 1
Server 2
P0
P3
P1
P2
C1
C2
C3
C4
C5
C6
Consumer Group A
Consumer Group B

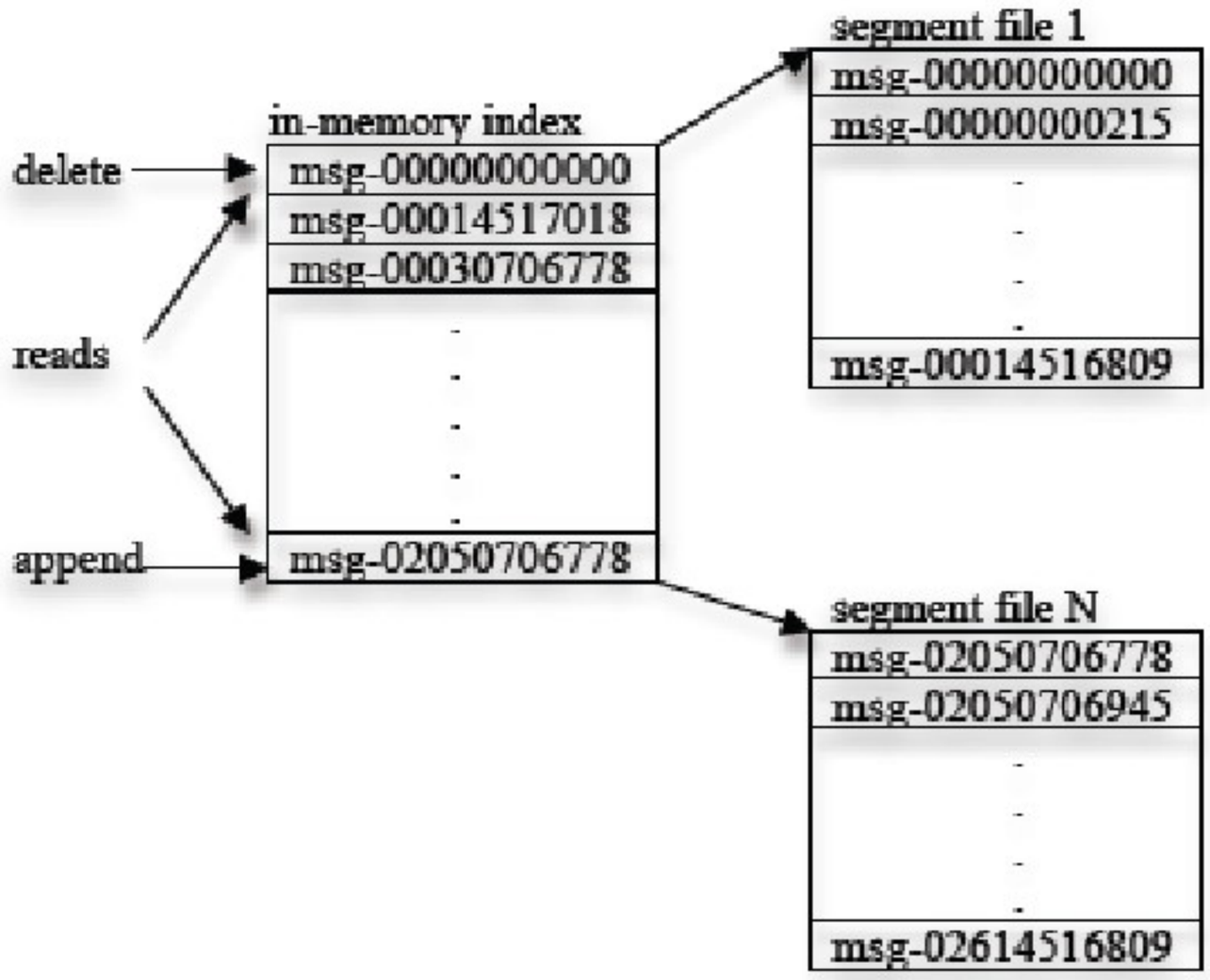
Kafka构造及原理

一、 持久化

- Kafka的存储布局非常简单。话题的每个分区对应一个逻辑日志。物理上，一个日志为相同大小的一组分段文件。每次生产者发布消息到一个分区，代理就将消息追加到最后一个段文件中。当发布的消息数量达到设定值或者经过一定的时间后，段文件真正flush磁盘中。写入完成后，消息公开给消费者。
- 与传统的消息系统不同，Kafka系统中存储的消息没有明确的消息Id。
- 消息通过日志中的逻辑偏移量来公开。这样就避免了维护配套的密集寻址，用于映射消息ID到实际消息地址的随机存取索引结构的开销。消息ID是增量的，但不连续。要计算下一消息的ID，可以在其逻辑偏移的基础上加上当前消息的长度。
- 消费者始终从特定分区顺序地获取消息，如果消费者知道特定消息的偏移量，也就说明消费者已经消费了之前的所有消息。消费者向代理发出异步拉请求，准备字节缓冲区用于消费。每个异步拉请求都包含要消费的消息偏移量

Kafka构造及原理

一、 持久化



Kafka构造及原理

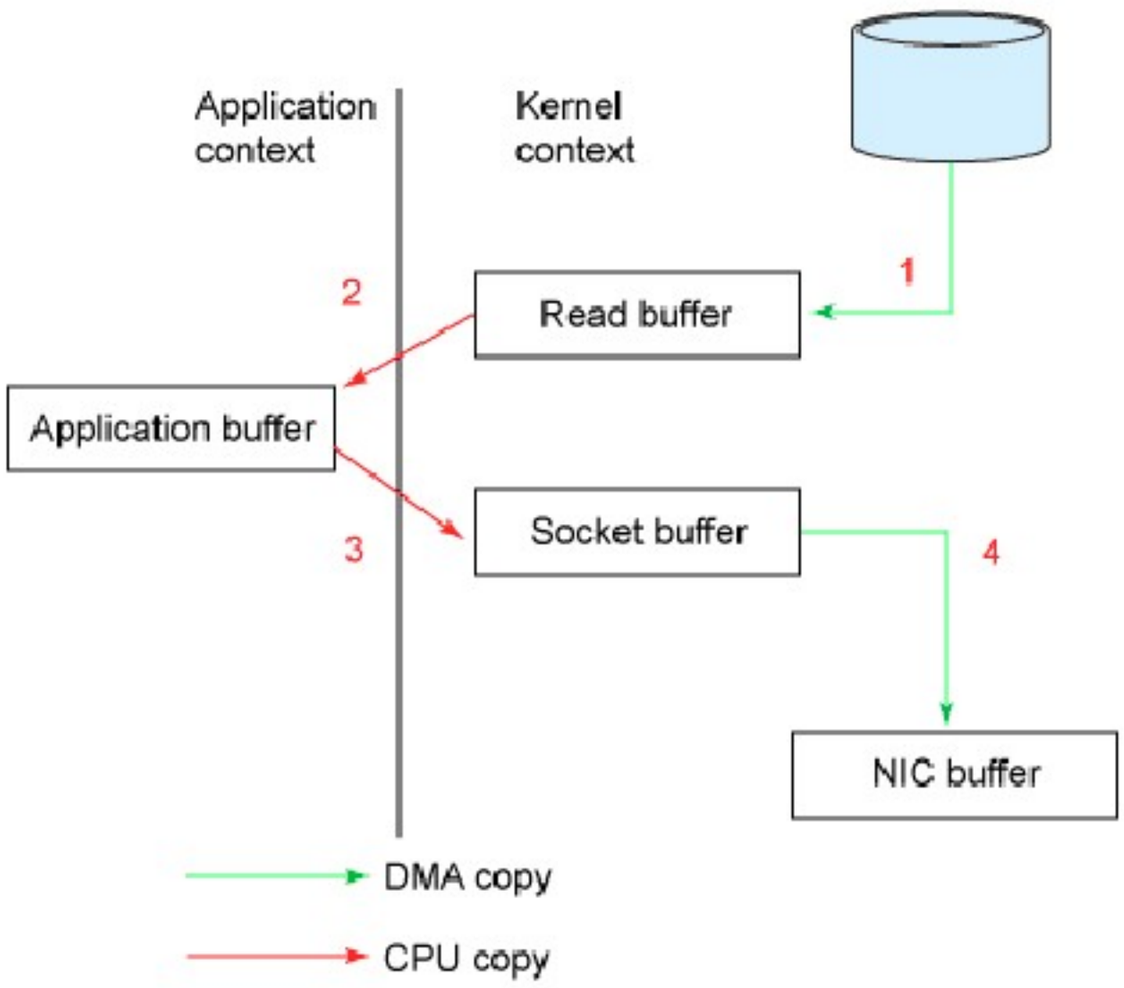
二、 传输效率

- 生产者提交一批消息作为一个请求。消费者虽然利用api遍历消息是一个一个的，但背后也是一次请求获取一批数据，从而减少网络请求数量。
- Kafka层采用无缓存设计，而是依赖于底层的文件系统页缓存。这有助于避免双重缓存，及即消息只缓存了一份在页缓存中。同时这在kafka重启后保持缓存warm也有额外的优势。因kafka根本不缓存消息在进程中，故gc开销也就很小。
- zero-copy：kafka为了减少字节拷贝，采用了大多数系统都会提供的sendfile系统调用。如下图：

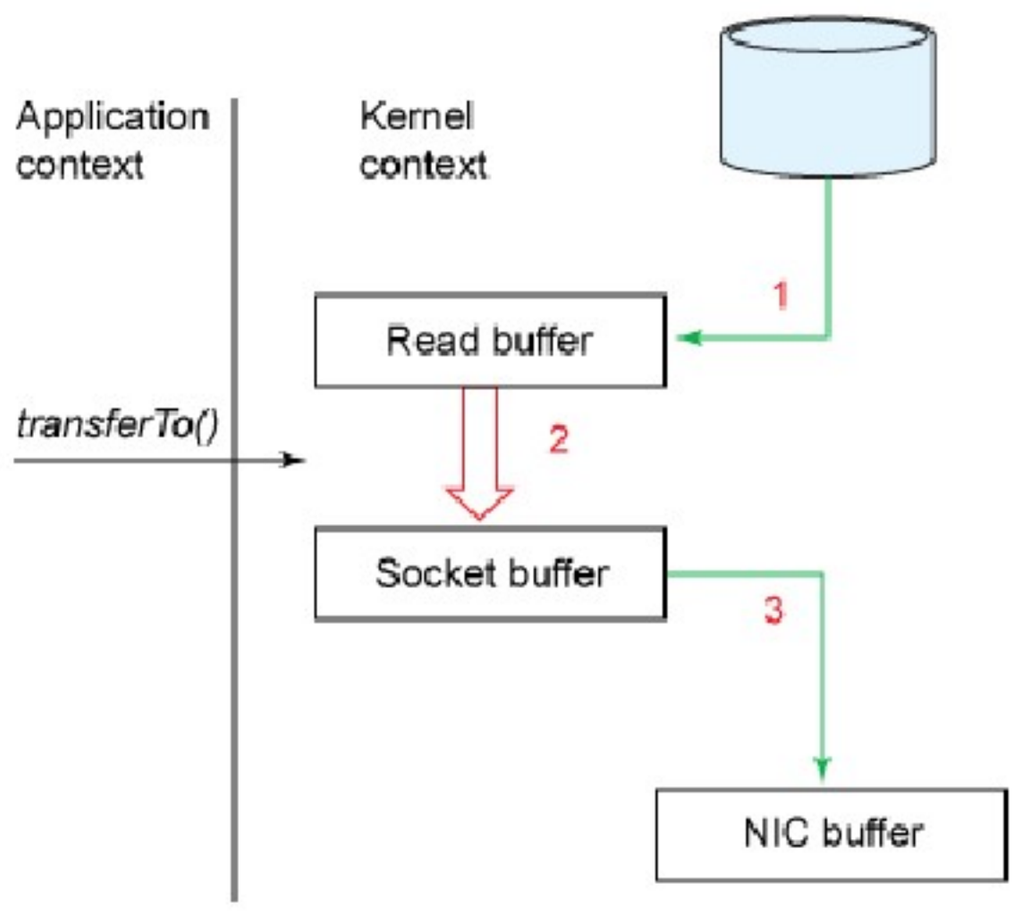
Kafka构造及原理

二、 传输效率

传统方式



zero-copy



Kafka构造及原理

三、 无状态的broker

与其它消息系统不同，Kafka代理是无状态的。这意味着消费者必须维护已消费的状态信息。这些信息由消费者自己维护，代理完全不管。这种设计非常微妙，它本身包含了创新。

- 从代理删除消息变得很棘手，因为代理并不知道消费者是否已经使用了该消息。Kafka创新性地解决了这个问题，它将一个简单的基于时间的SLA应用于保留策略。当消息在代理中超过一定时间后，将会被自动删除。
- 这种创新设计有很大的好处，消费者可以故意倒回到老的偏移量再次消费数据。这违反了队列的常见约定，但被证明是许多消费者的基本特征。

Kafka构造及原理

三、 交付保证

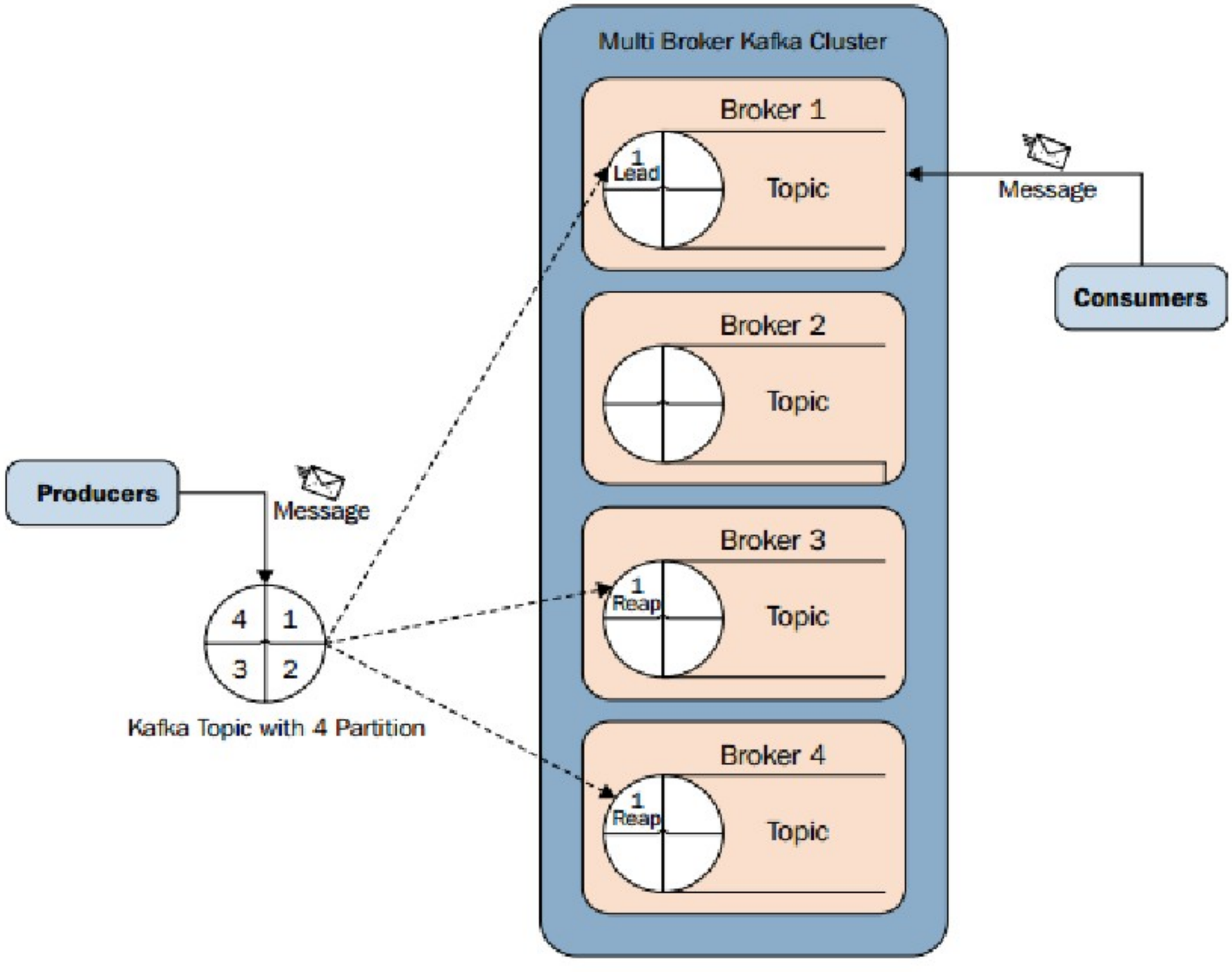
- Kafka默认采用at least once的消息投递策略。即在消费者端的处理顺序是获得消息->处理消息->保存位置。这可能导致一旦客户端挂掉，新的客户端接管时处理前面客户端已处理过的消息。

Kafka构造及原理

四、 副本管理

- kafka将日志复制到指定多个服务器上。
- 复本的单元是partition。在正常情况下，每个分区有一个leader和0到多个follower。
- leader处理对应分区上所有的读写请求。分区可以多于broker数，leader也是分布式的。
- follower的日志和leader的日志是相同的， ollower被动的复制leader。如果leader挂了，其中一个follower会自动变成新的leader.

Kafka构造及原理


Multi Broker Kafka Cluster
Broker 1
1
Lead
Topic
Broker 2
Topic
Broker 3
1
Reap
Topic
Broker 4
1
Reap
Topic
Message
Consumers
Producers
Message
4
1
3
2
Kafka Topic with 4 Partition

Kafka构造及原理

四、 副本管理

- 和其他分布式系统一样，节点“活着”的定义在于我们能否处理一些失败情况。

  kafka需要两个条件保证是“活着”

  1.节点在**zookeeper**注册的**session**还在且可维护（基于**zookeeper**心跳机制）

  2.如果是**slave**则能够紧随**leader**的更新不至于落得太远。

- **kafka**采用**in sync**来代替“活着”。如果**follower**挂掉或卡住或落得很远，则**leader**会移除同步列表中的**in sync**。至于落了多远才叫远由**replica.lag.max.messages**配置，而表示复本“卡住”由**replica.lag.time.max.ms**配置。

Kafka构造及原理

四、 副本管理

- 所谓一条消息是“提交”的，意味着所有in sync的复本也持久化到了他们的log中。这意味着消费者无需担心leader挂掉导致数据丢失。另一方面，生产者可以选择是否等待消息“提交”。
- kafka动态的维护了一组in-sync(ISR)的复本，表示已追上了leader,只有处于该状态的成员组才是能被选择为leader。这些ISR组会在发生变化时被持久化到zookeeper中。通过ISR模型和f+1复本，可以让kafka的topic支持最多f个节点挂掉而不会导致提交的数据丢失。

Kafka构造及原理

五、 分布式协调

- 由于kafka中一个topic中的不同分区只能被消费组中的一个消费者消费，就避免了多个消费者消费相同的分区时会导致额外的开销（如要协调哪个消费者消费哪个消息，还有锁及状态的开销）。kafka中消费进程只需要在代理和同组消费者有变化时时进行一次协调（这种协调不是经常性的，故可以忽略开销）。
- kafka使用zookeeper做以下事情：

  1.探测broker和consumer的添加或移除

  2.当1发生时触发每个消费者进程的重新负载。

  3.维护消费关系和追踪消费者在分区消费的消息的offset。

Zookeeper的使用

- **Broker Node Registry**

  /brokers/ids/[0...N] --> host:port (ephemeral node)

  - broker启动时在/brokers/ids下创建一个znode，把broker id写进去。
  - 因为broker把自己注册到zookeeper中实用的是瞬时节点，所以这个注册是动态的，如果broker宕机或者没有响应该节点就会被删除。

- **Broker Topic Registry**

  /brokers/topics/[topic]/[0...N] --> nPartions (ephemeral node)

  每个broker把自己存储和维护的partion信息注册到该路径下。

Zookeeper的使用

- **Consumers and Consumer Groups**
  - consumers也把它们自己注册到zookeeper上，用以保持消费负载平衡和offset记录。
  - group id相同的多个consumer构成一个消费租，共同消费一个topic，同一个组的consumer会尽量均匀的消费，其中的一个consumer只会消费一个partion的数据。
- **Consumer Id Registry**

/consumers/[group_id]/ids/[consumer_id] --> {"topic1": #streams, ..., "topicN": #streams} (ephemeral node)

  - 每个consumer在/consumers/[group_id]/ids下创建一个瞬时的唯一的consumer_id，用来描述当前该group下有哪些consumer是alive的，如果消费进程挂掉对应的consumer_id就会从该节点删除。

Zookeeper的使用

- **Consumer Offset Tracking**

  /consumers/[group_id]/offsets/[topic]/[partition_id] --> offset_counter_value ((persistent node)

  consumer把每个partition的消费offset记录保存在该节点下。

- **Partition Owner registry**

  /consumers/[group_id]/owners/[topic]/[broker_id-partition_id] --> consumer_node_id (ephemeral node)

  该节点维护着partion与consumer之间的对应关系。

Zookeeper的使用

- **Consumer registration algorithm**

1. Register itself in the consumer id registry under its group.
2. Register a watch on changes (new consumers joining or any existing consumers leaving) under the consumer id registry. (Each change triggers rebalancing among all consumers within the group to which the changed consumer belongs.)
3. Register a watch on changes (new brokers joining or any existing brokers leaving) under the broker id registry. (Each change triggers rebalancing among all consumers in all consumer groups.)
4. If the consumer creates a message stream using a topic filter, it also registers a watch on changes (new topics being added) under the broker topic registry. (Each change will trigger re-evaluation of the available topics to determine which topics are allowed by the topic filter. A new allowed topic will trigger rebalancing among all consumers within the consumer group.)
5. Force itself to rebalance within in its consumer group.

Zookeeper的使用

- **Consumer rebalancing algorithm**

Each consumer does the following during rebalancing:

1. For each topic T that C_i subscribes to
2. let P_T be all partitions producing topic T
3. let C_G be all consumers in the same group as C_i that consume topic T
4. sort P_T (so partitions on the same broker are clustered together)
5. sort C_G
6. let i be the index position of C_i in C_G and let N = size(P_T)/size(C_G)
7. assign partitions from i*N to (i+1)*N - 1 to consumer C_i
8. remove current entries owned by C_i from the partition owner registry
9. add newly assigned partitions to the partition owner registry
   (we may need to re-try this until the original partition owner releases its ownership)

Apache Kafka对比其它消息服务

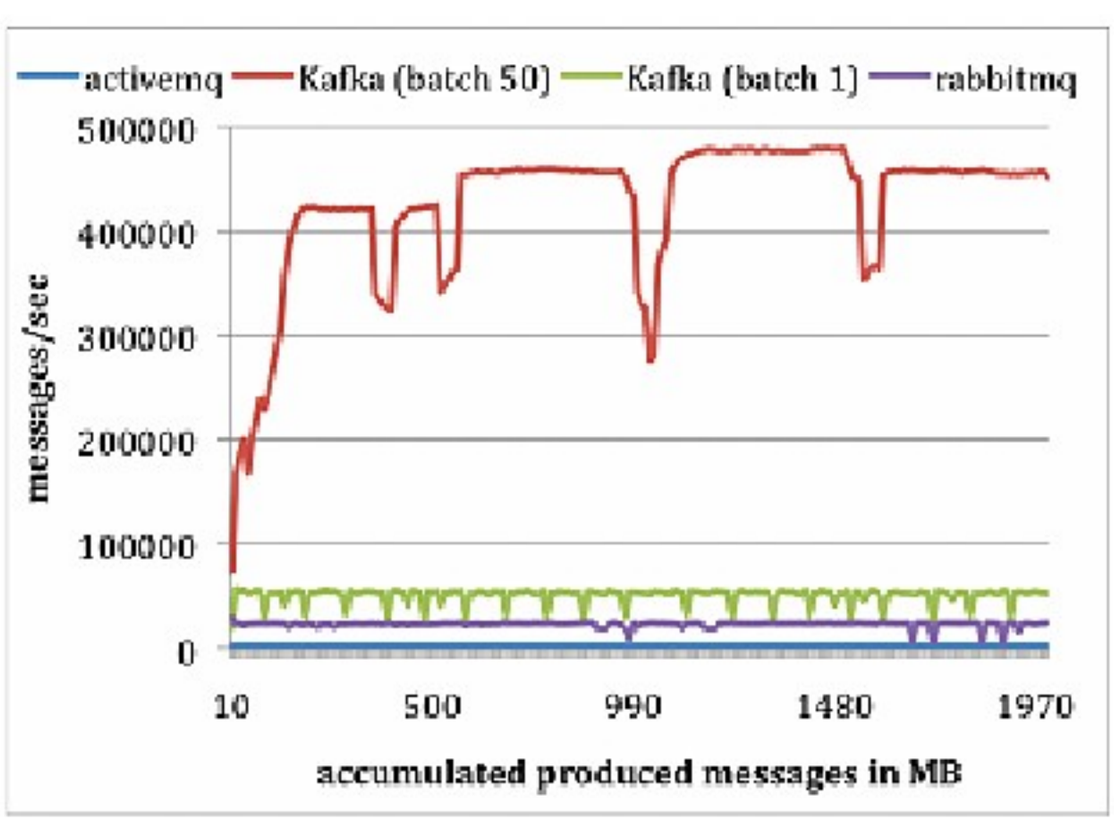
- **测试环境**

  LinkedIn团队做了个实验研究，对比Kafka与Apache ActiveMQ V5.4和RabbitMQ V2.4的性能。他们使用ActiveMQ默认的消息持久化库Kahadb。LinkedIn在两台Linux机器上运行他们的实验，每台机器的配置为8核2GHz、16GB内存，6个磁盘使用RAID10。两台机器通过1GB网络连接。一台机器作为代理，另一台作为生产者或者消费者。

Apache Kafka对比其它消息服务

- **生产者测试**

  对每个系统，运行一个生产者，总共发布1000万条消息，每条消息200字节。Kafka生产者以1和50批量方式发送消息。ActiveMQ和RabbitMQ似乎没有简单的办法来批量发送消息，LinkedIn假定它的批量值为1。结果如下图所示：



Kafka性能要好很多的主要原因包括：

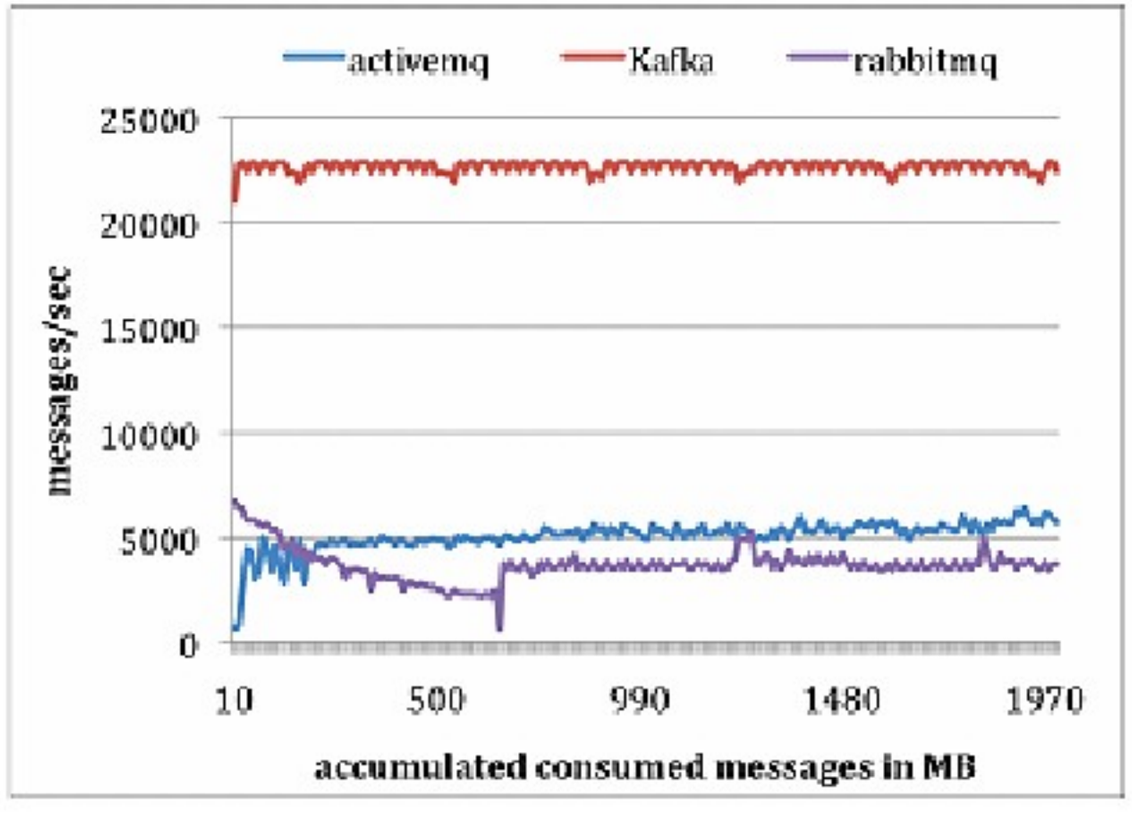
1. Kafka不等待代理的确认，以代理能处理的最快速度发送消息。

2. Kafka有更高效的存储格式。平均而言，Kafka每条消息有9字节的开销，而ActiveMQ有144字节。其原因是JMS所需的沉重消息头，以及维护各种索引结构的开销。LinkedIn注意到ActiveMQ一个最忙的线程大部分时间都在存取B-Tree以维护消息元数据和状态。

Apache Kafka对比其它消息服务

- **消费者测试**

  为了做消费者测试，LinkedIn使用一个消费者获取总共1000万条消息。LinkedIn让所有系统每次读取请求都预获取大约相同数量的数据，最多1000条消息或者200KB。对ActiveMQ和RabbitMQ，LinkedIn设置消费者确认模型为自动。结果如图所示。



Kafka性能要好很多的主要原因包括：

1. Kafka有更高效的存储格式，在Kafka中，从代理传输到消费者的字节更少。

2. ActiveMQ和RabbitMQ两个容器中的代理必须维护每个消息的传输状态。LinkedIn团队注意到其中一个ActiveMQ线程在测试过程中，一直在将KahaDB页写入磁盘。与此相反，Kafka代理没有磁盘写入动作。最后，Kafka通过使用sendfile API降低了传输开销