消息中间件KAFKA

一个高性能、分布式的消息系统

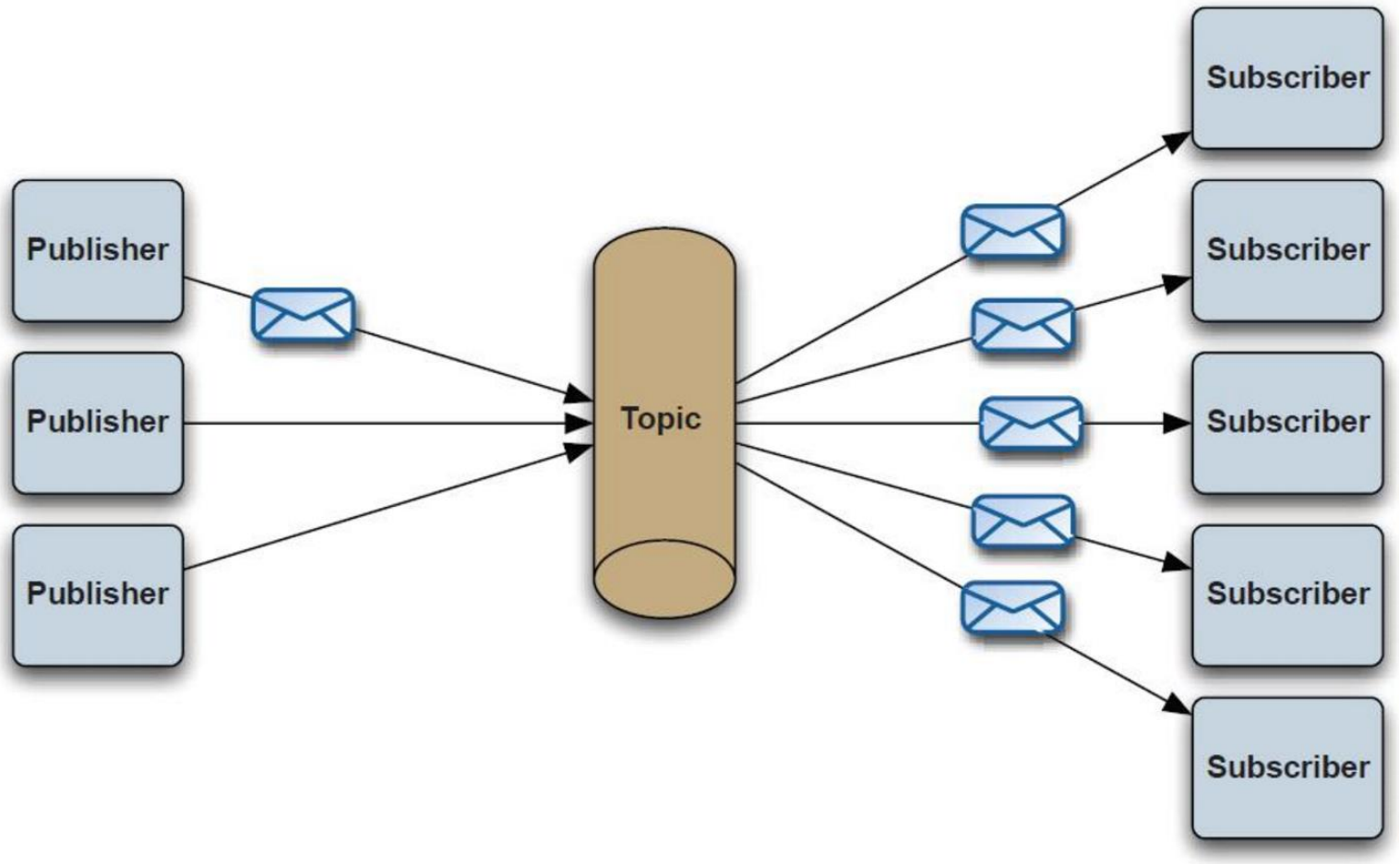
artoderk@gmail.com

目录

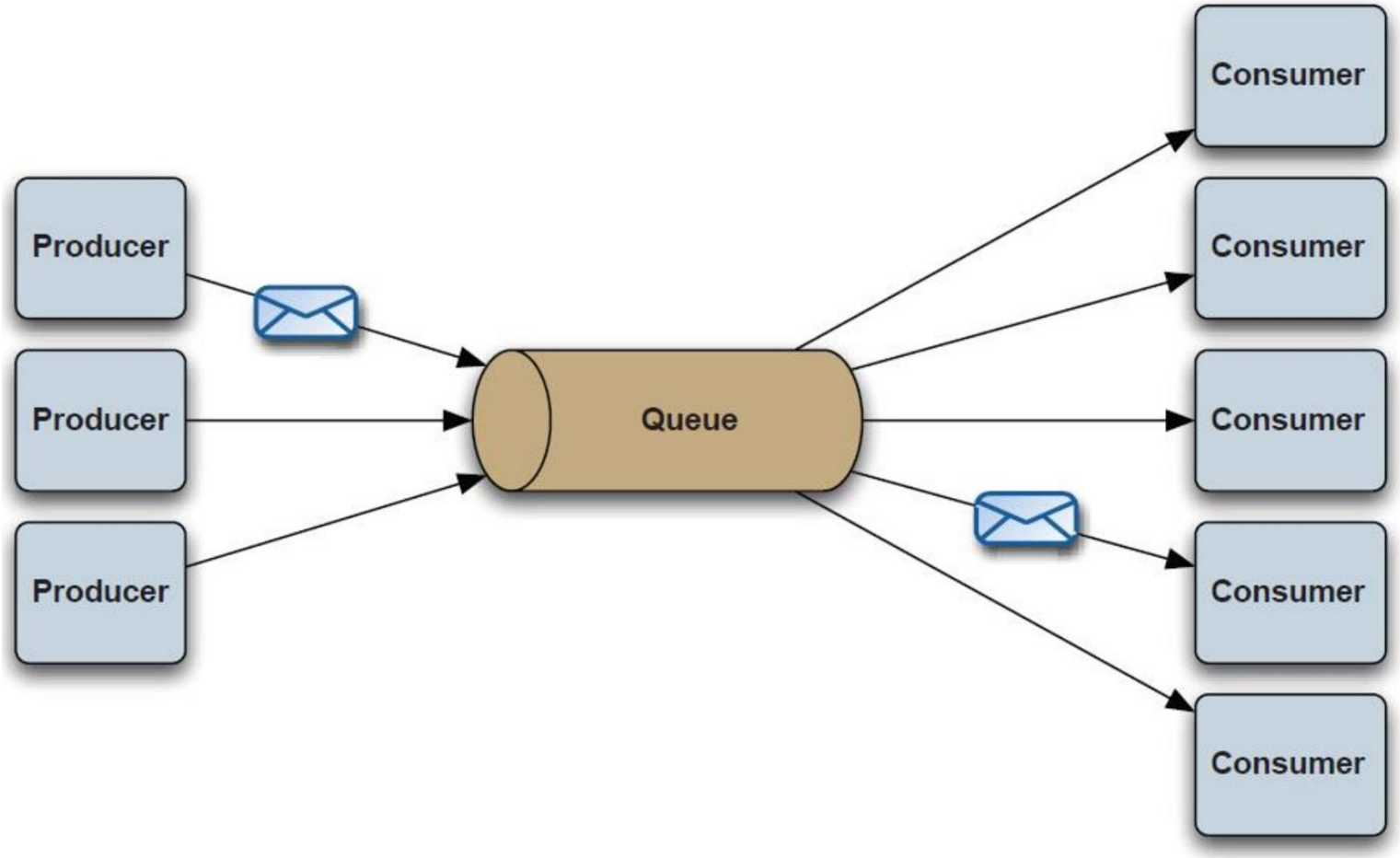
消息中间件

- 什么是消息中间件
- 为什么使用消息中间件
- 常用的消息中间件
- 消息的基本通信模式
  - 发布/订阅模式(publish-subscribe)
  - 点对点模式(P2P)

发布/订阅模式(publish-subscribe)



点对点模式(P2P)


Producer
Producer
Producer
Queue
Consumer
Consumer
Consumer
Consumer
Consumer

Kafka介绍

- Kafka最初是由Linkedin公司开发，使用Scala语言编写，运行于JVM上，使用Zookeeper协调管理的一个分布式的消息系统。Linkedin在2010年贡献给了Apache基金会，成为Apache的顶级项目。

特点

- 消息持久化的时间复杂度为O(1) 。
- 高吞吐率。
- 支持消息分区、保证每个分区内的消息有序。
- Scale out：支持在线水平扩展。

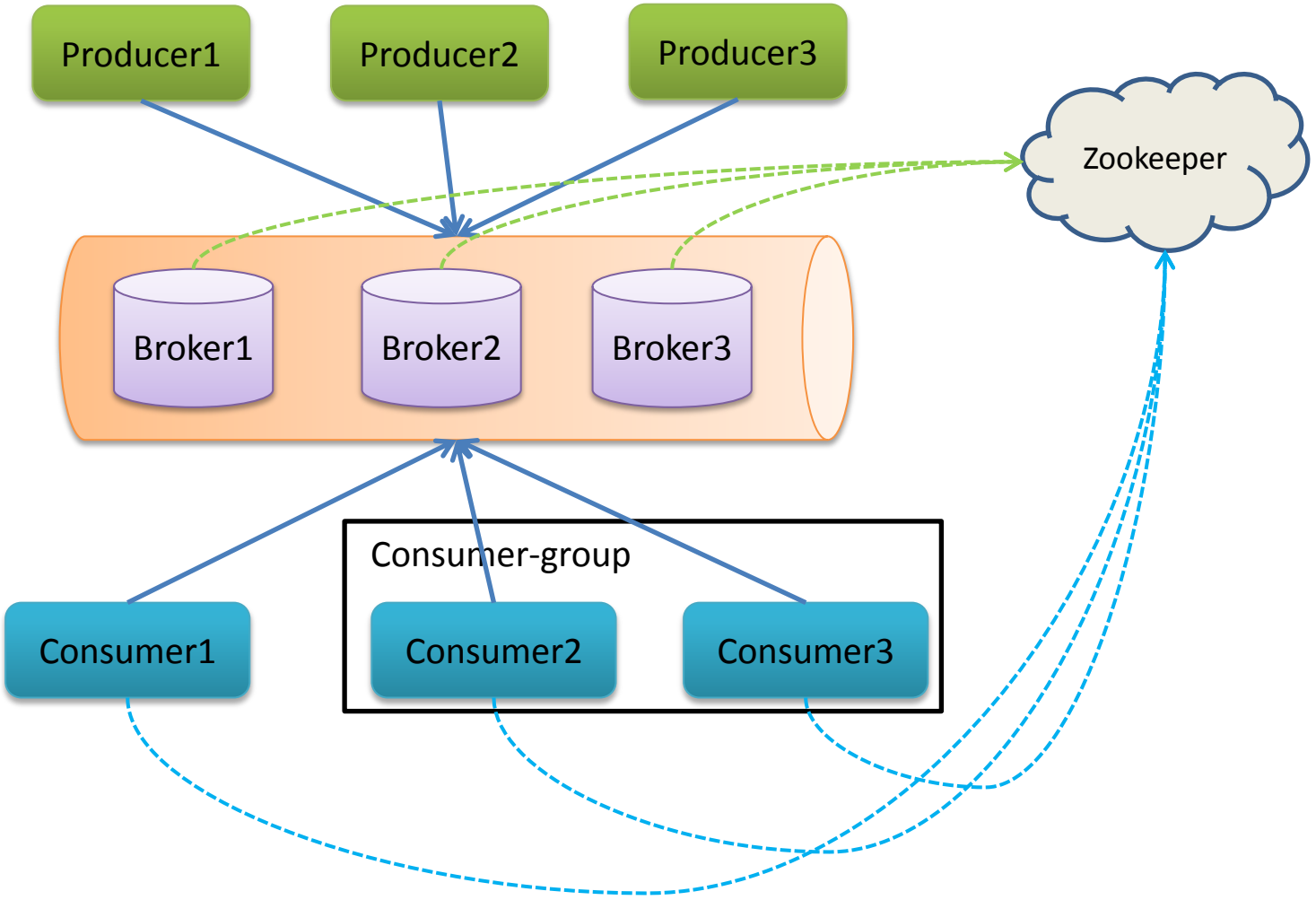
Kafka架构

- 名词介绍
- 架构图

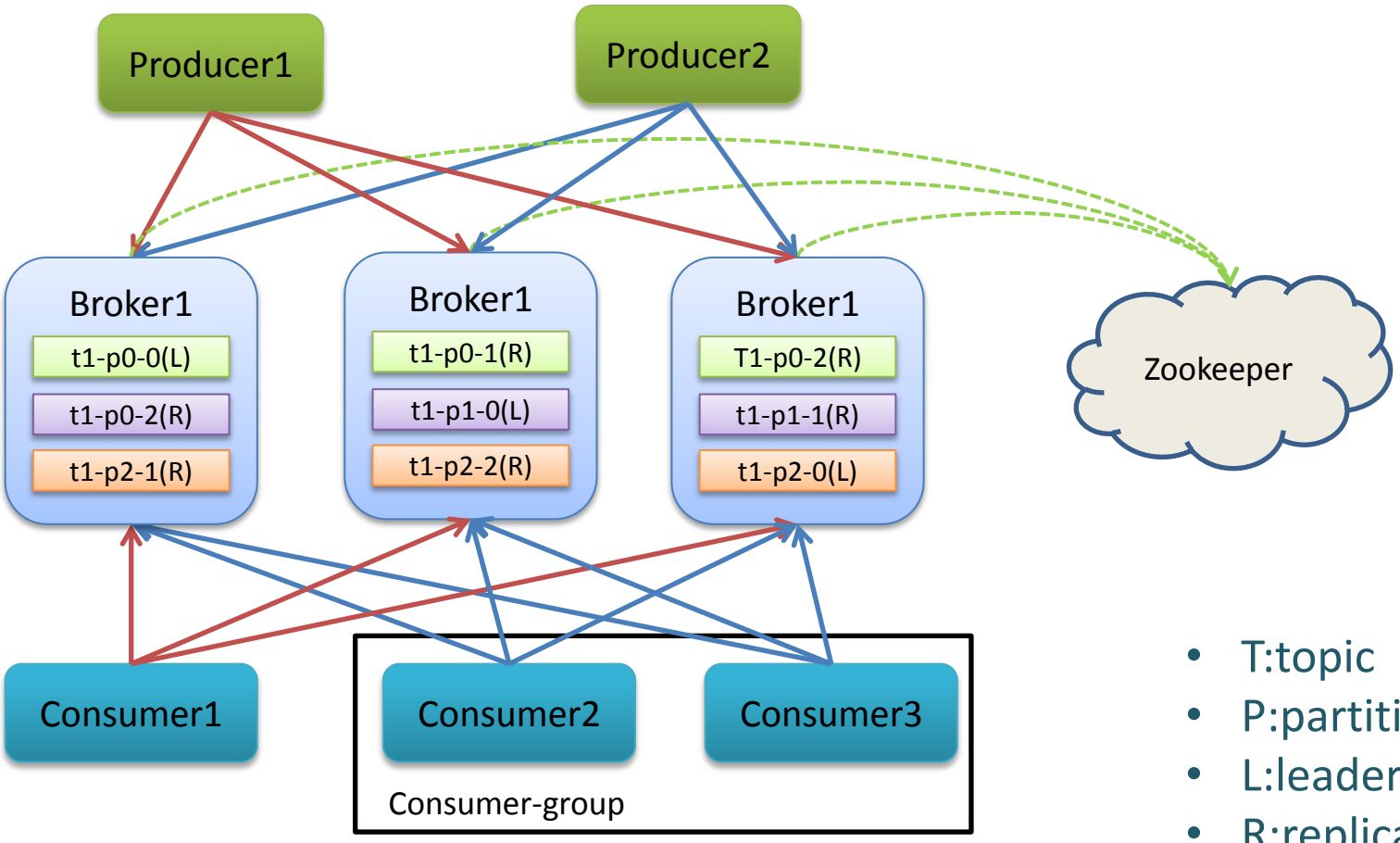
名词介绍

- **Broker**
  Kafka集群包含一个或多个服务器，这种服务器被称为broker
- **Topic**
  每条发布到Kafka上消息都有一个类别，这个类别被称为Topic
- **Partition**
  Parition是物理上的概念，每个Topic包含一个或多个Partition
- **Producer**
  负责发布消息到Kafka broker
- **Consumer**
  消息消费者，向Kafka broker读取消息的客户端
- **Consumer Group**
  每个Consumer属于一个特定的Consumer Group（可为每个Consumer指定group name，若不指定group name则属于默认的group）

架构图1


Producer1
Producer2
Producer3
Zookeeper
Broker1
Broker2
Broker3
Consumer-group
Consumer1
Consumer2
Consumer3

架构图2



- T:topic
- P:partition
- L:leader
- R:replication

生产者

- 生产者使用方式
- 生产者发送流程
- 生产者负载均衡

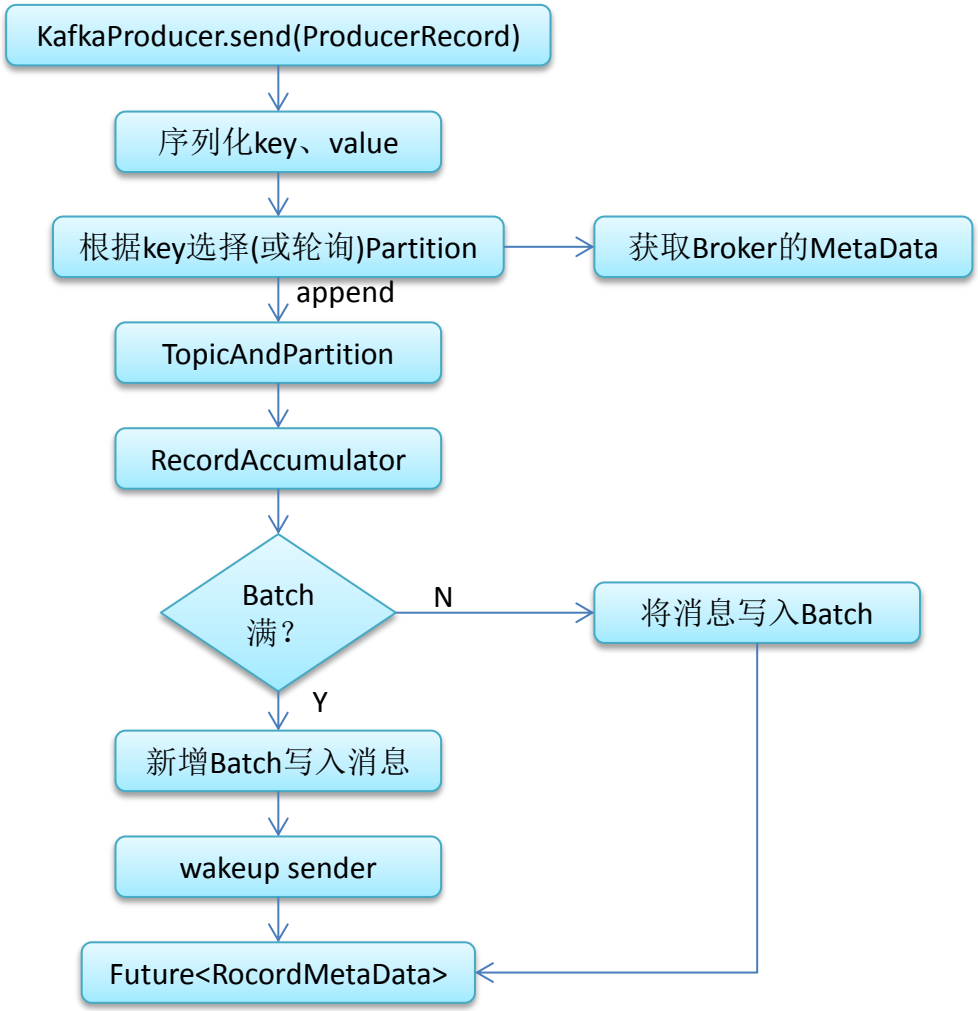
生产者使用方式

- 发送方式(同步、异步)
- 确认方式(acks=0、 acks=1、 acks=all/-1)

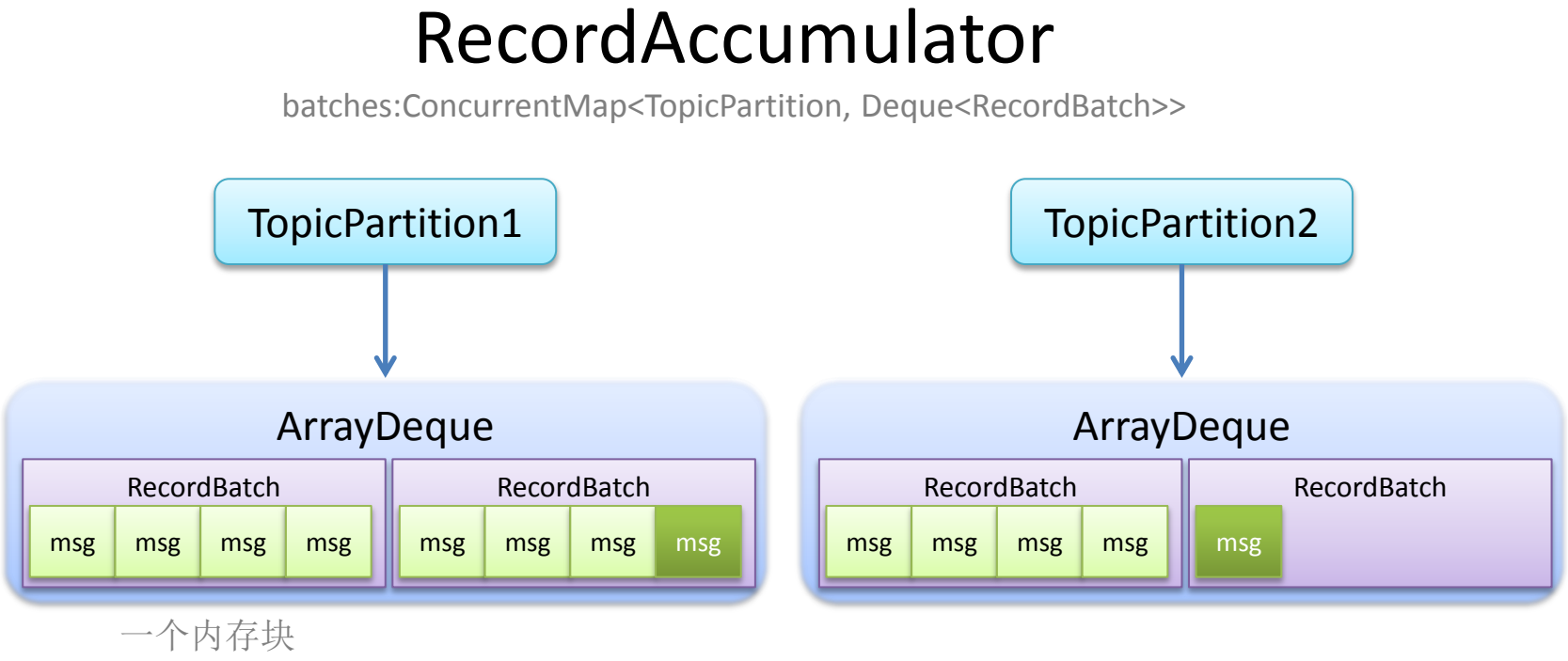
```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:4242");
props.put("acks", "all");
props.put("retries", 0);
props.put("batch.size", 16384);
props.put("linger.ms", 1);
props.put("buffer.memory", 33554432);
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

org.apache.kafka.clients.producer.Producer<String, String> producer = new KafkaProducer<>(props);
for(int i = 0; i < 100; i++) {
// producer.send(new ProducerRecord<String, String>("my-topic", Integer.toString(i), Integer.toString(i)));
   producer.send(new ProducerRecord<String, String>("my-topic", Integer.toString(i)), new Callback() {
       @Override
       public void onCompletion(RecordMetadata arg0, Exception arg1) {
           // TODO Auto-generated method stub
           System.out.println(arg0.offset());
       }
   });
}
producer.close();
```

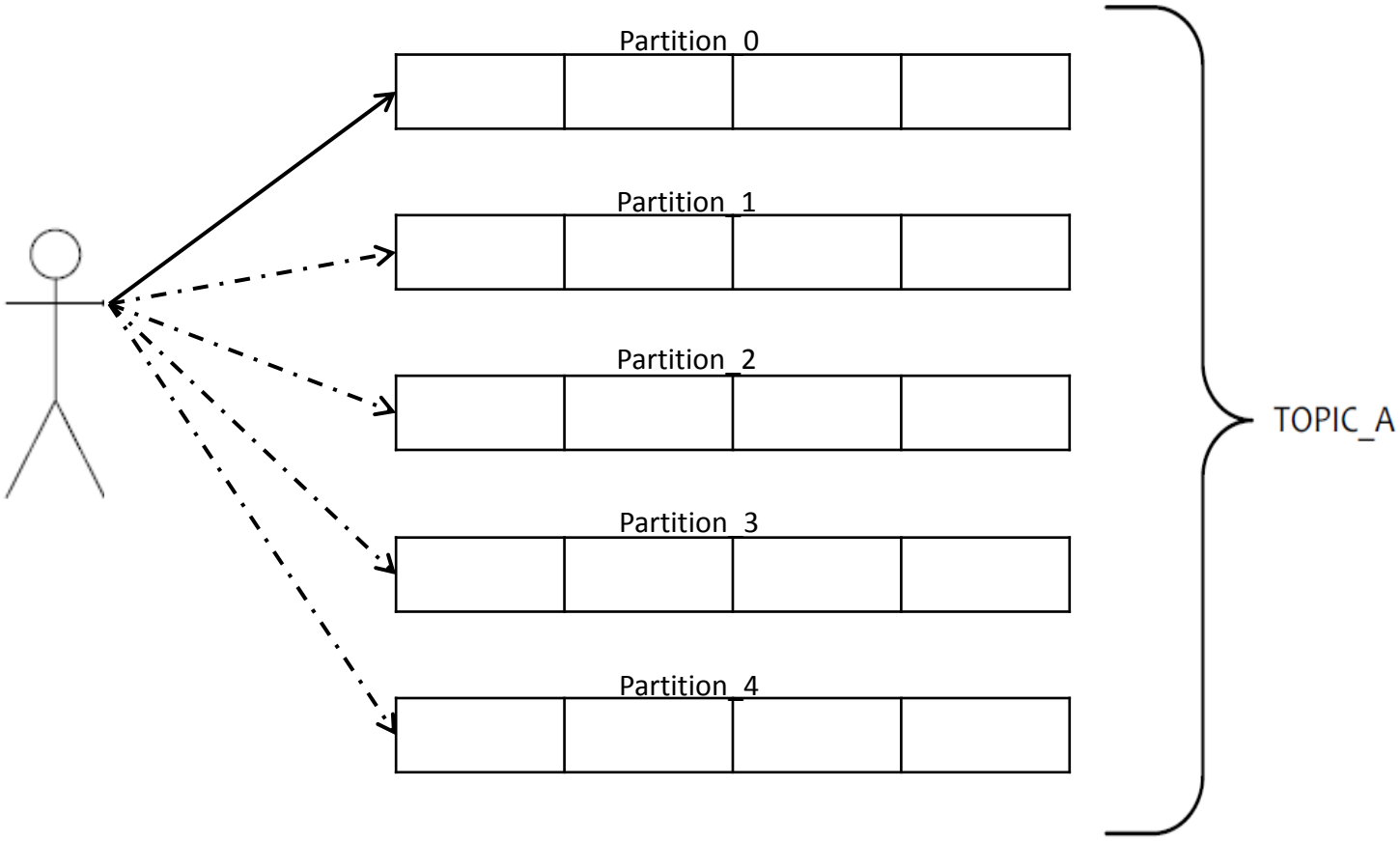
生产者发送流程1


KafkaProducer.send(ProducerRecord)
序列化key、value
根据key选择(或轮询)Partition
获取Broker的MetaData
append
TopicAndPartition
RecordAccumulator
Batch
满？
N
将消息写入Batch
Y
新增Batch写入消息
wakeup sender
Future<RocordMetaData>

生产者发送流程2


RecordAccumulator
batches:ConcurrentMap<TopicPartition, Deque<RecordBatch>>
TopicPartition1
TopicPartition2
ArrayDeque
RecordBatch
msg
msg
msg
msg
RecordBatch
msg
msg
msg
msg
ArrayDeque
RecordBatch
msg
msg
msg
msg
RecordBatch
msg
一个内存块

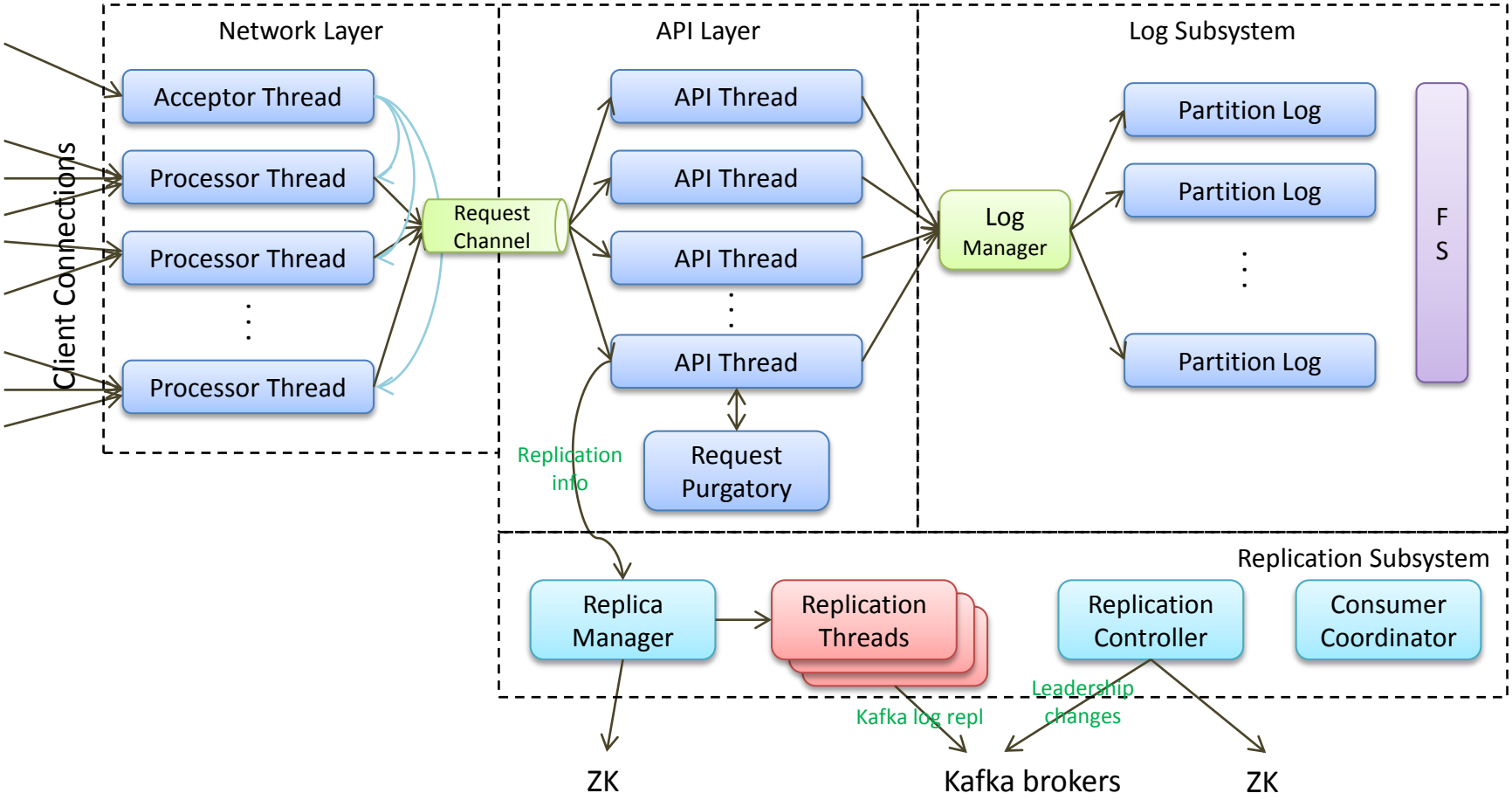
生产者负载均衡


Partition_0
Partition_1
Partition_2
Partition_3
Partition_4
TOPIC_A

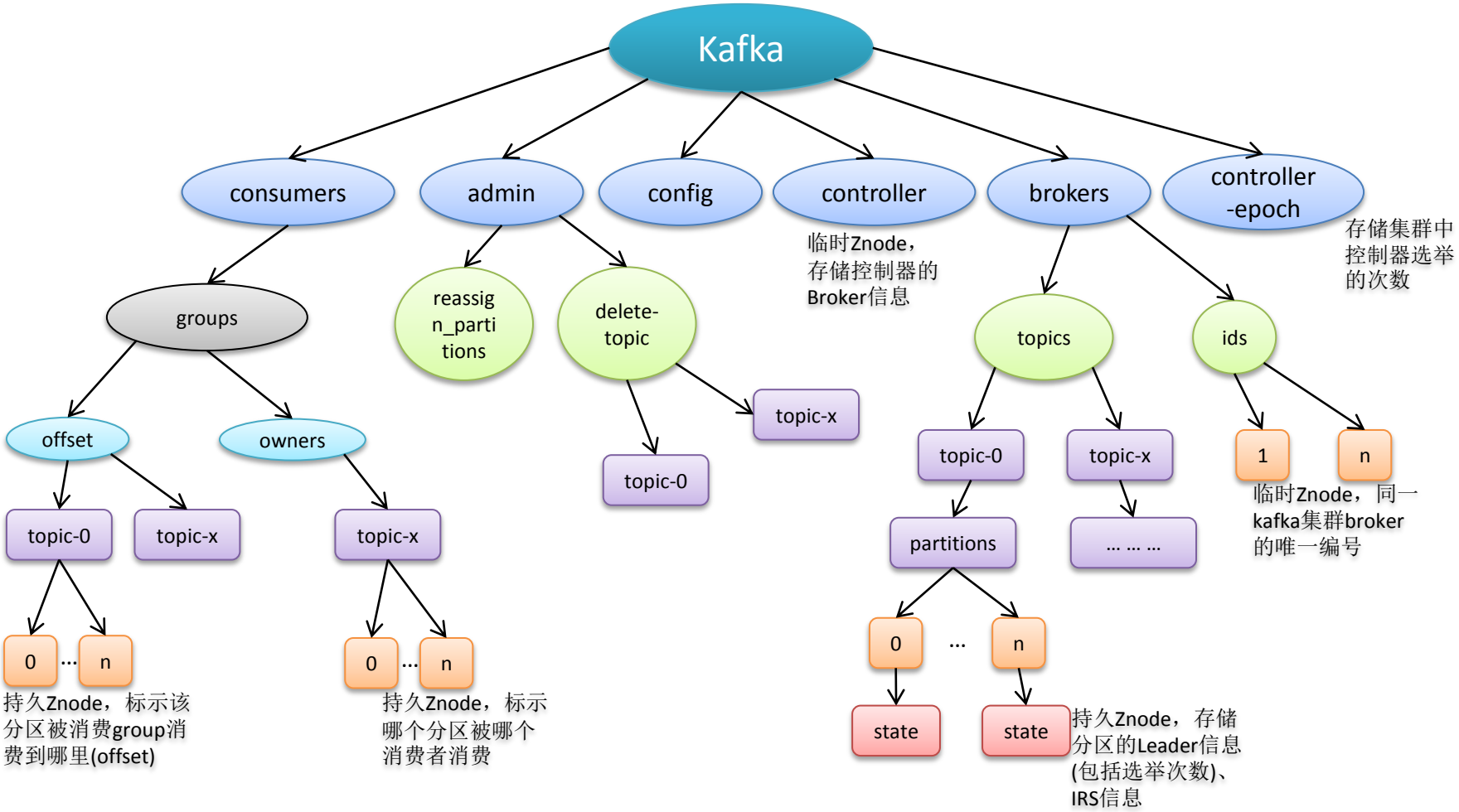
服务端

- 服务端模块结构
- ZK目录结构
- 控制器(Controller)
- 消息的持久化与复制
- 协调器(Coordinator)

服务端模块结构


Network Layer
API Layer
Log Subsystem
Client Connections
Acceptor Thread
Processor Thread
Processor Thread
Processor Thread
Request Channel
API Thread
API Thread
API Thread
API Thread
Log Manager
Partition Log
Partition Log
Partition Log
F S
Replication info
Request Purgatory
Replication Subsystem
Replica Manager
Replication Threads
Replication Controller
Consumer Coordinator
Kafka log repl
Leadership changes
ZK
Kafka brokers
ZK

Zk目录结构


Kafka
consumers
admin
config
controller
brokers
controller -epoch
存储集群中控制器选举的次数
临时Znode，存储控制器的Broker信息
groups
reassig n_parti tions
delete-topic
topics
ids
offset
owners
topic-0
topic-x
topic-0
topic-x
topic-0
topic-x
1
n
临时Znode，同一kafka集群broker的唯一编号
partitions
... ...
0
...
n
0
...
n
state
state
持久Znode，存储分区的Leader信息(包括选举次数)、IRS信息
持久Znode，标示该分区被消费group消费到哪里(offset)
持久Znode，标示哪个分区被哪个消费者消费

Controller

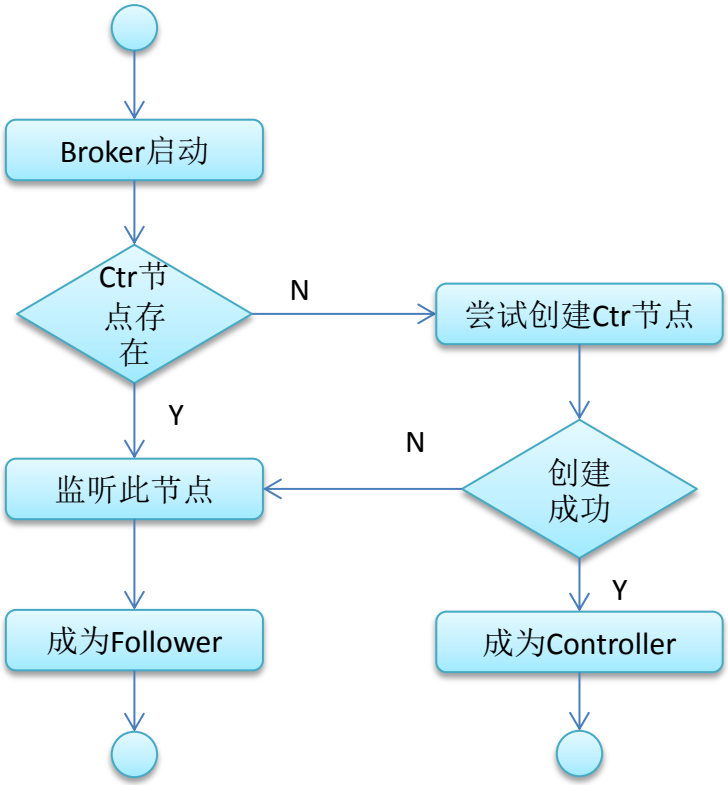
- Controller介绍
- 启动流程
- 创建Topic
- Broker宕机

Controller介绍

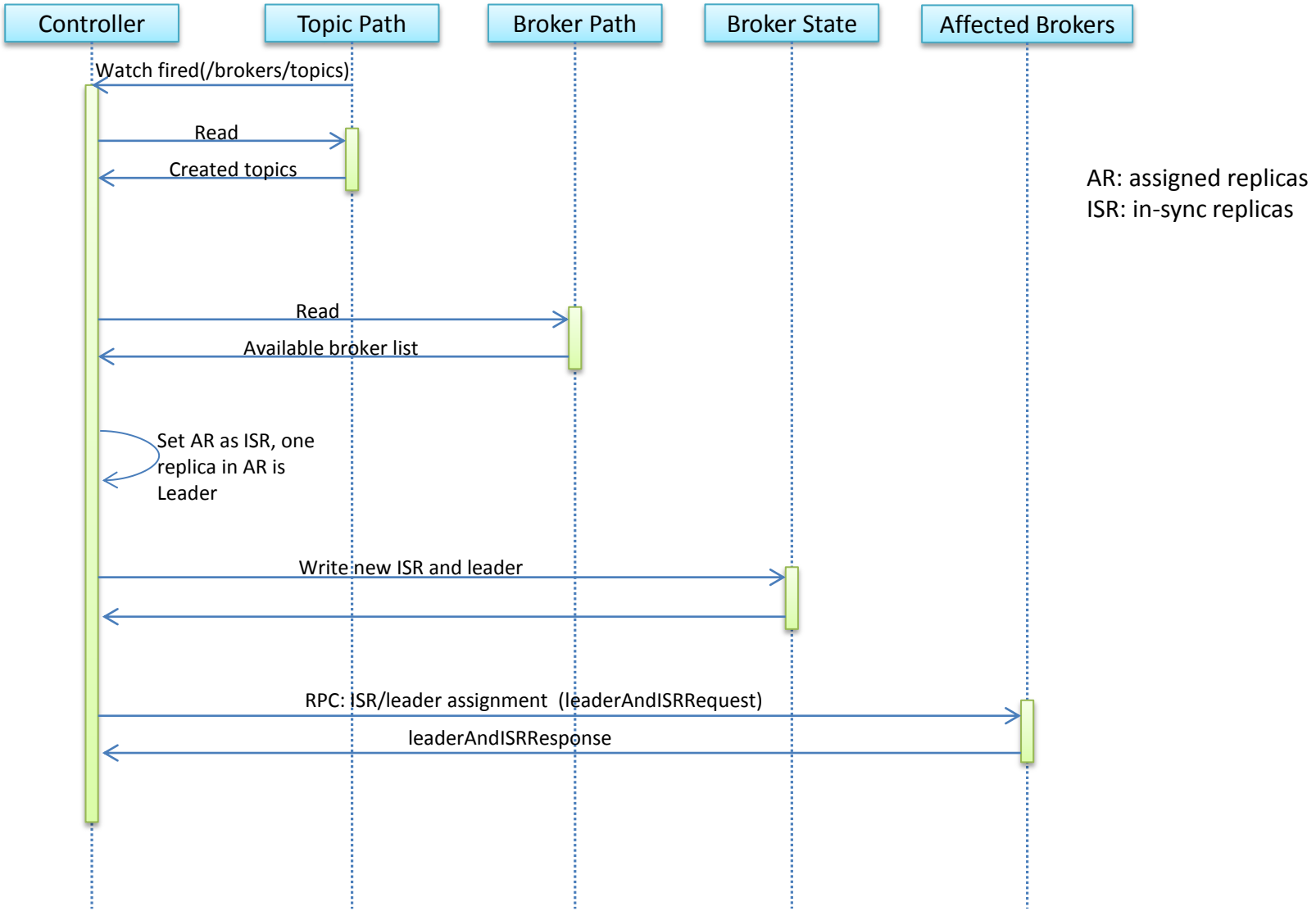
Controller是为了replica机制而创建的，负责:

- Partition的Leader选举
- 增删Topic
- Replica的重新分配

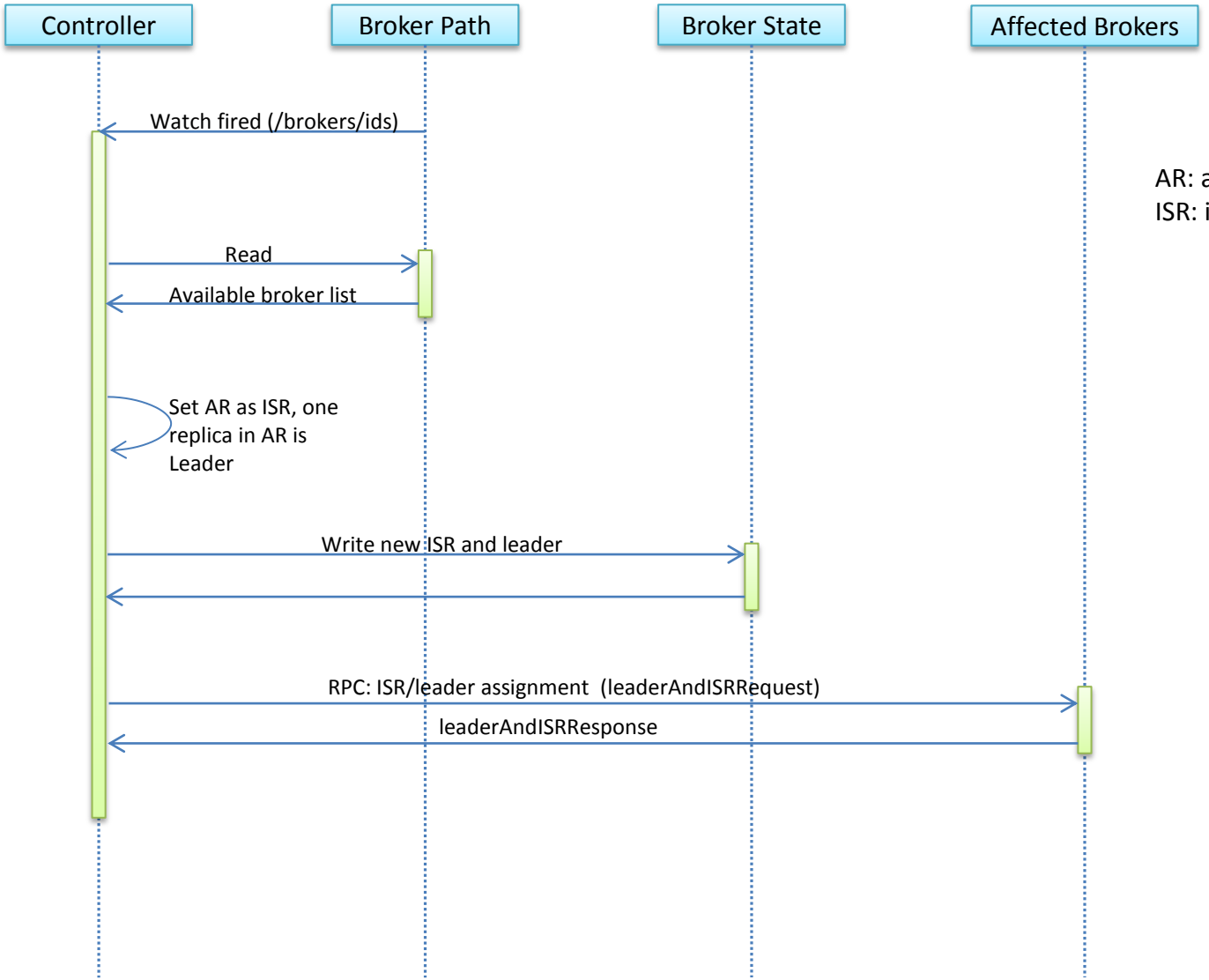
Controller启动流程



创建Topic


Controller
Topic Path
Broker Path
Broker State
Affected Brokers
Watch fired(/brokers/topics)
Read
Created topics
Read
Available broker list
Set AR as ISR, one replica in AR is Leader
Write new ISR and leader
RPC: ISR/leader assignment (leaderAndISRRequest)
leaderAndISRResponse
AR: assigned replicas
ISR: in-sync replicas

Broker宕机



AR: assigned replicas
ISR: in-sync replicas

消息的持久化与复制

- Kafka日志介绍
- Replication介绍

Kafka日志介绍1

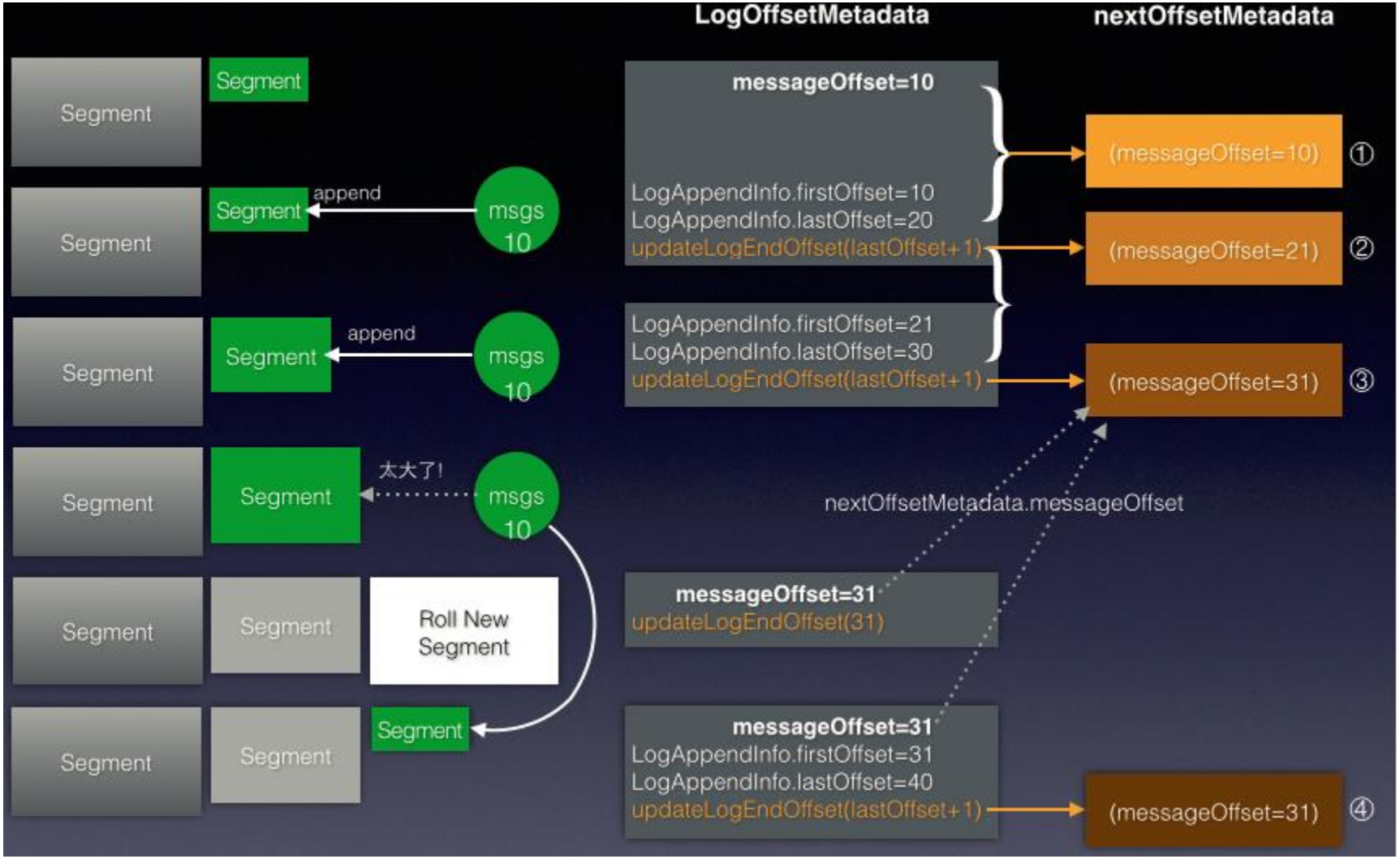
Kafka的消息都存储在日志系统中

假如kafka集群中只有一个broker，数据文件目录为message-folder，例如创建一个topic名称为：report_push，partitions=4，则存储路径和目录规则为：

xxx/message-folder

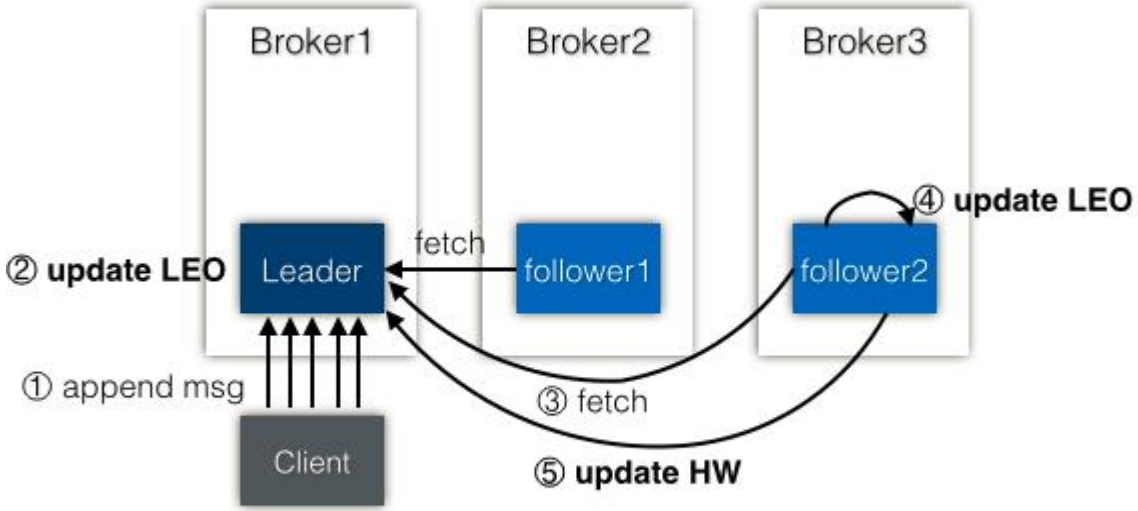
|--report_push-0
|--report_push-1
|--report_push-2
|--report_push-3

Kafka日志介绍2


LogOffsetMetadata
nextOffsetMetadata
Segment
Segment
Segment
append
msgs
10
Segment
Segment
append
msgs
10
Segment
Segment
太大了!
msgs
10
Segment
Segment
Roll New
Segment
Segment
Segment
Segment
messageOffset=10
LogAppendInfo.firstOffset=10
LogAppendInfo.lastOffset=20
updateLogEndOffset(lastOffset+1)
(messageOffset=10)
①
(messageOffset=21)
②
LogAppendInfo.firstOffset=21
LogAppendInfo.lastOffset=30
updateLogEndOffset(lastOffset+1)
(messageOffset=31)
③
nextOffsetMetadata.messageOffset
messageOffset=31
updateLogEndOffset(31)
messageOffset=31
LogAppendInfo.firstOffset=31
LogAppendInfo.lastOffset=40
updateLogEndOffset(lastOffset+1)
(messageOffset=31)
④

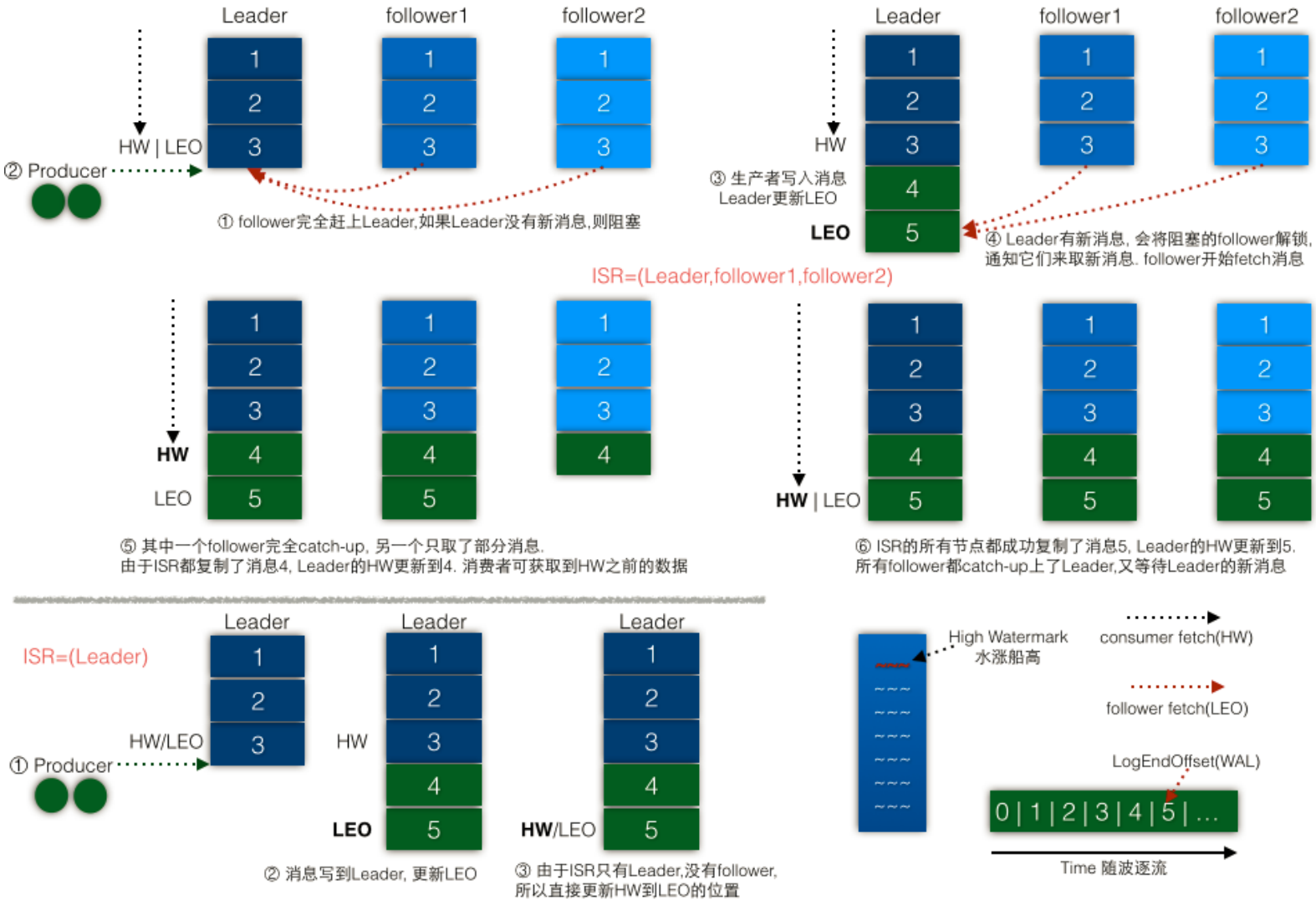
Replication介绍

每个Partition都可以有多个Replication，由 Leader副本负责读写, Follower副本负责从Leader拉取数据



HW: HighWatermark
LEO: LogEndOffset

In-Sync Replicas



Coordinator

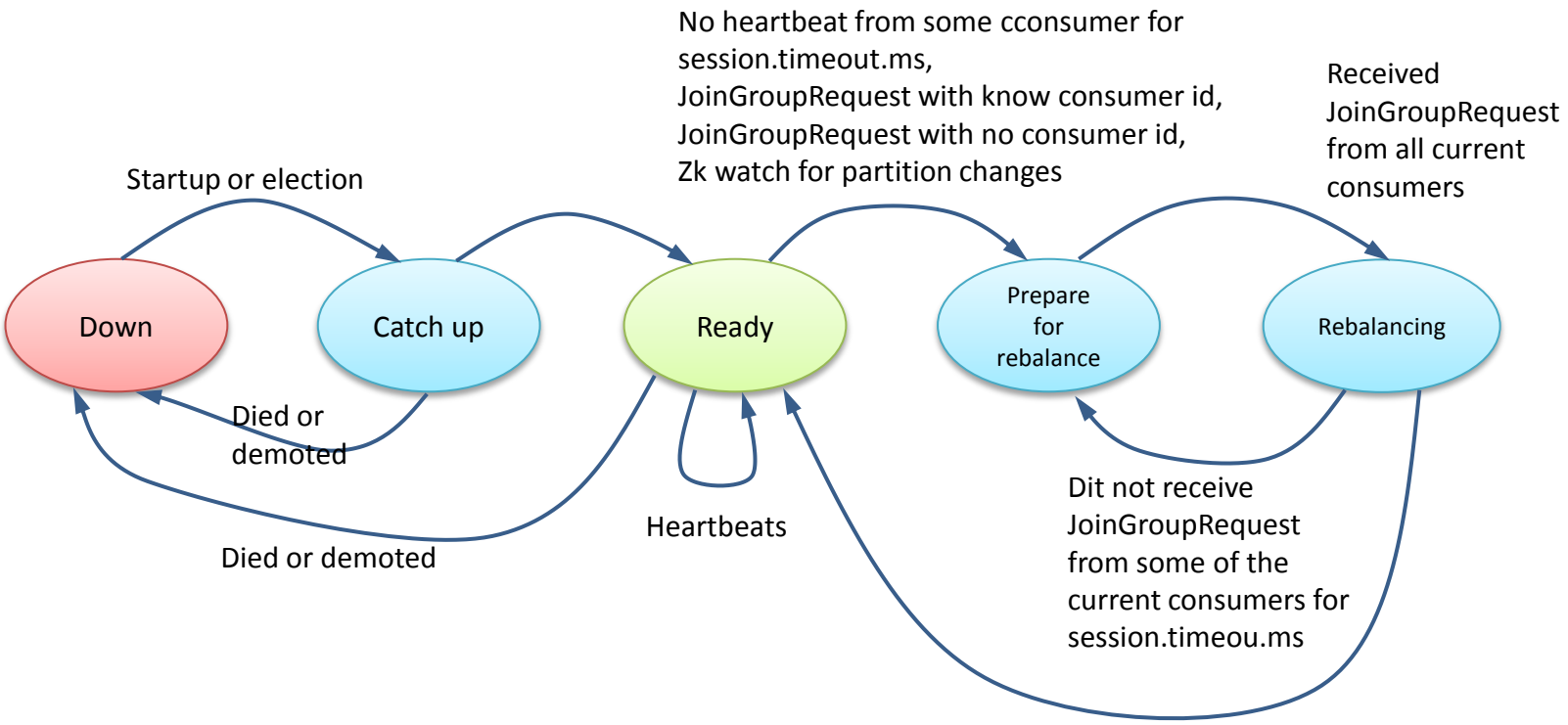
- Coordinator介绍
- Coordinator状态图
- Consumer负载均衡

Coordinator介绍

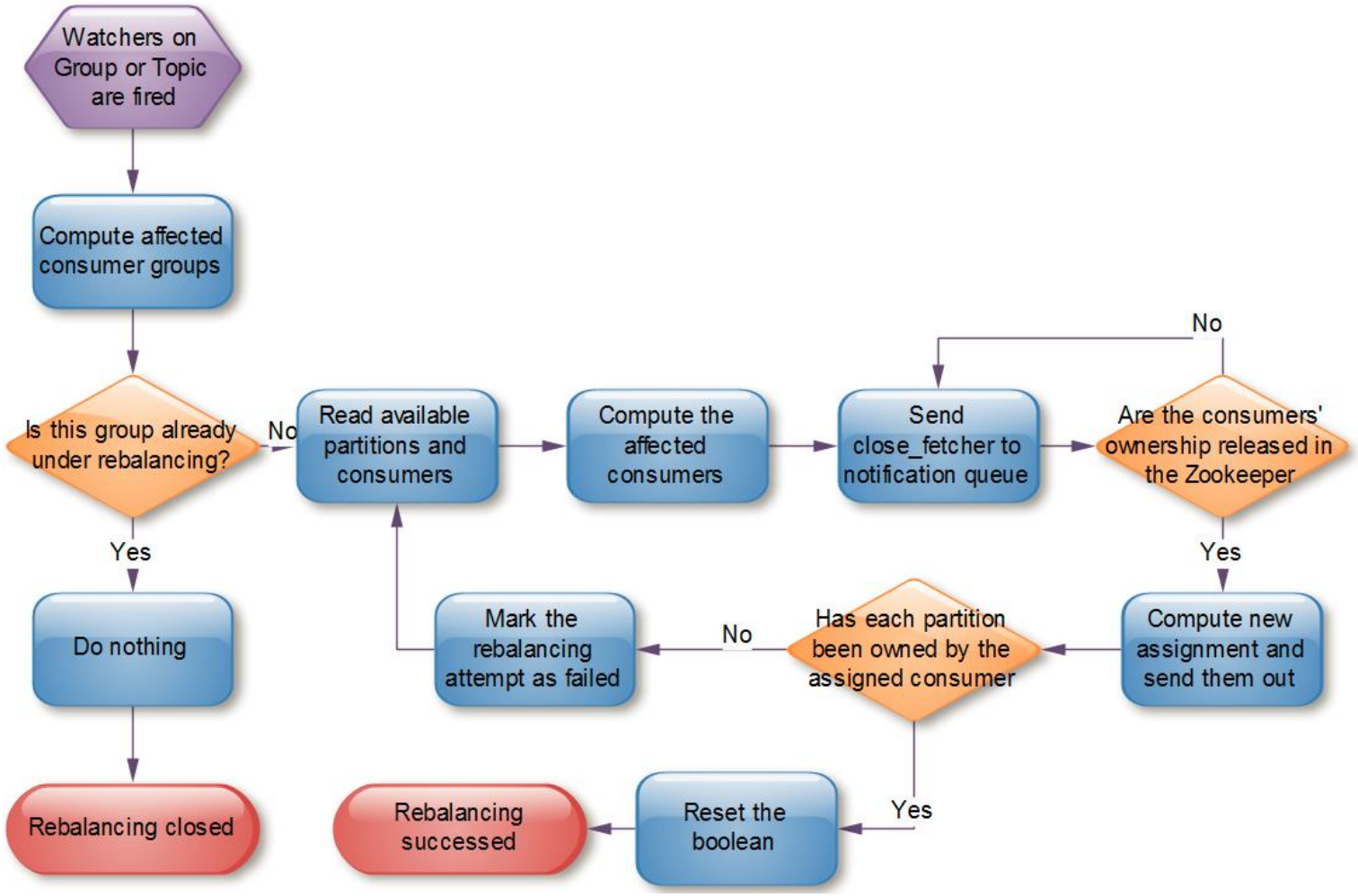
主要职责是管理Consumer的负载均衡:

- 给Consumer分配Partition
- Consumer故障检测
- Consumer再分配

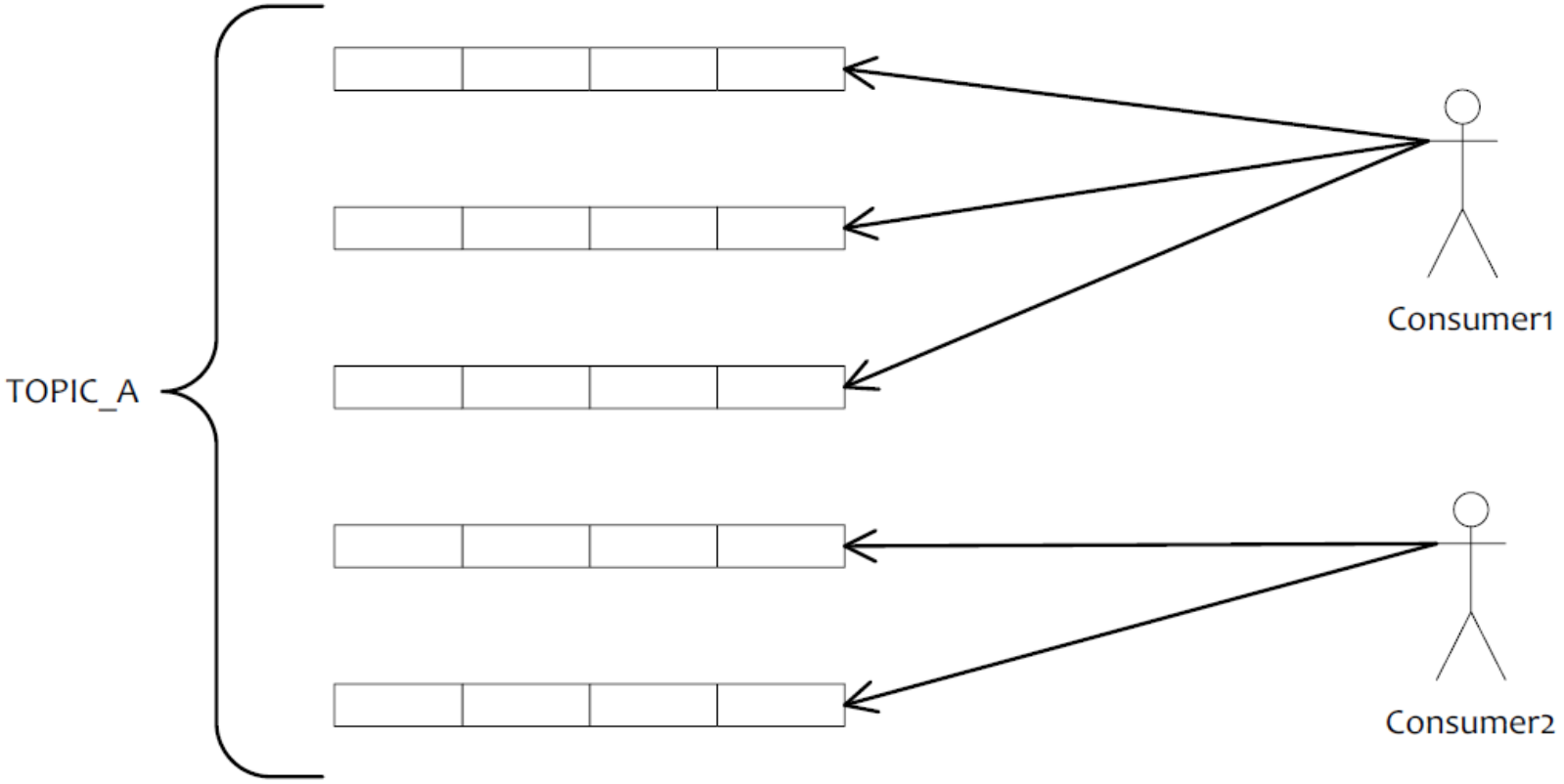
Coordinator状态图



Coordinator的Rebalance流程



Consumer负载均衡1



Consumer负载均衡2

队列数量	Consumer 数量	Rebalance 结果
5	2	C1: 3 C2: 2
6	3	C1: 2 C2: 2 C3: 2
10	20	C1~C10: 1 C11~C20: 0
20	6	C1: 4 C2: 4 c3~C6: 3

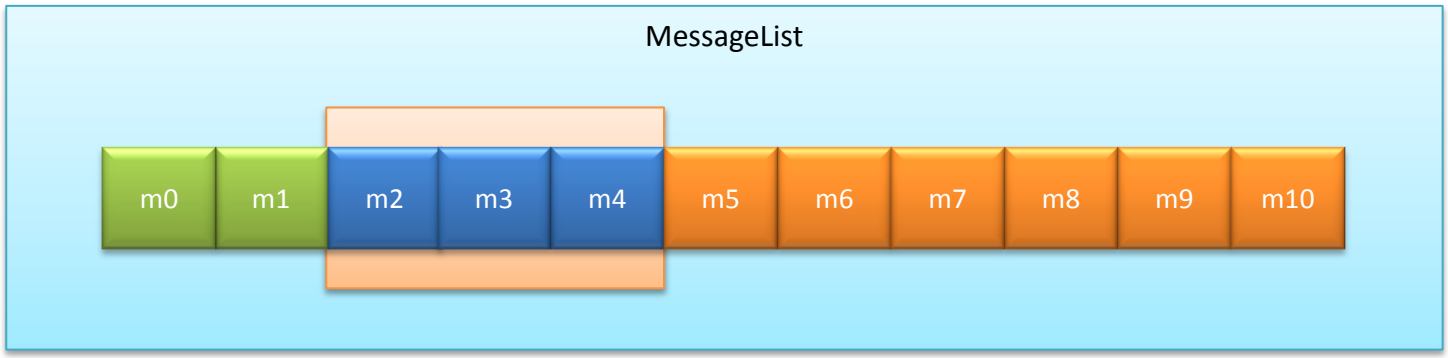
消费者

- 消息确认(Offset)
- 消费者使用方式
- 消费者启动流程

消息确认(Offset)

消费端Offset用来标示消息消费到哪个位置

- Offset存储方式(ZK、Broker、Custom)
- Offset提交方式(Autocommit、Custom)

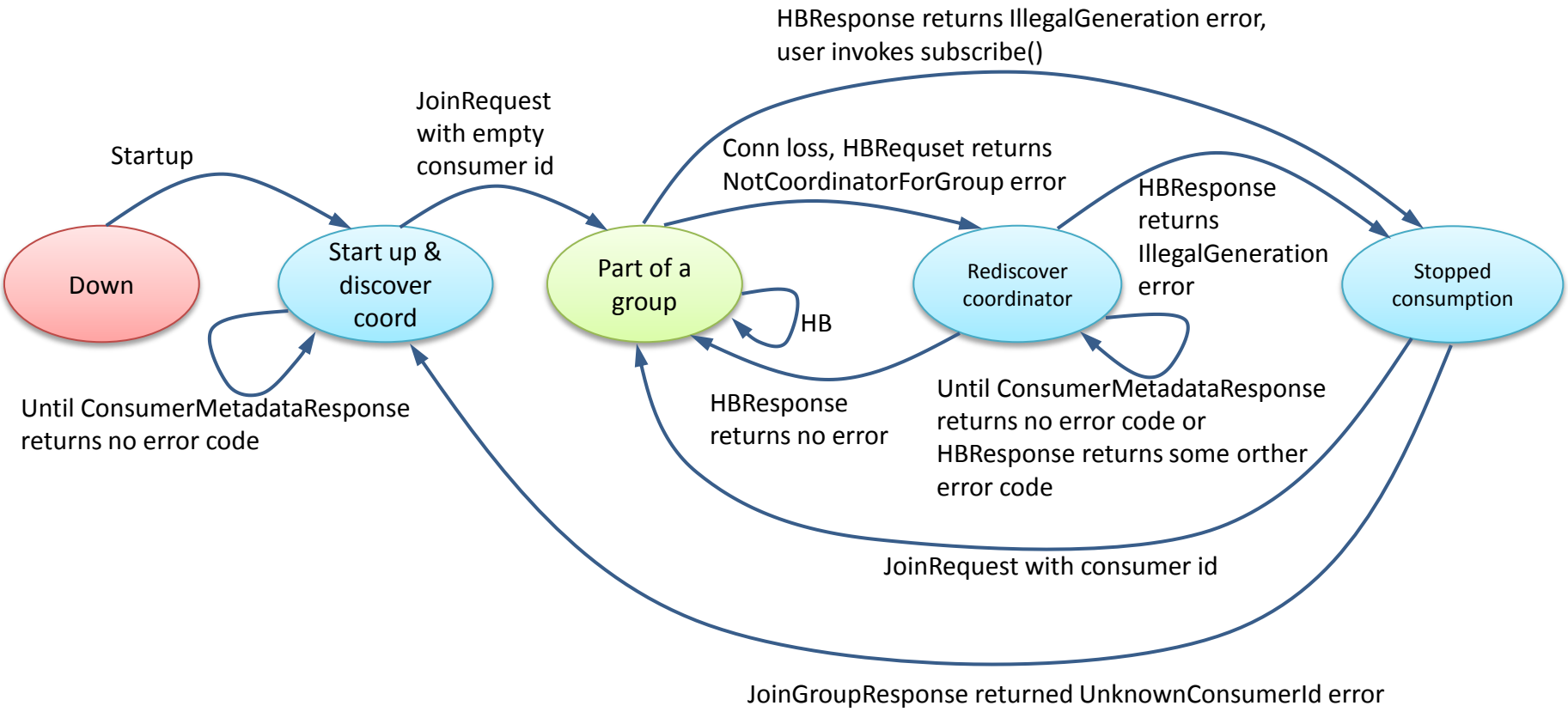


消费者使用方式

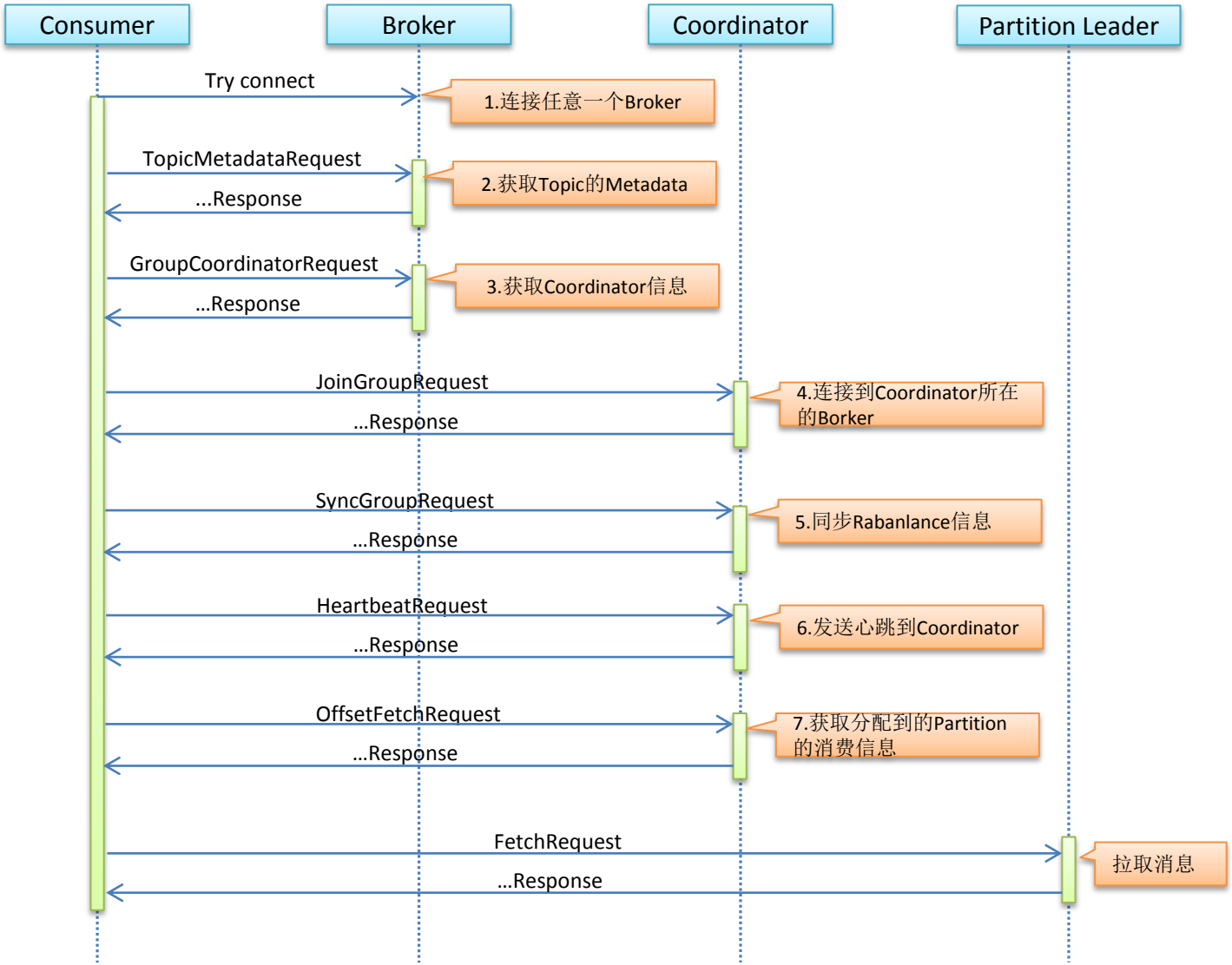
- Offset提交方式
- “fetch.message.max.bytes”需和服务端参数匹配

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "test");
props.put("enable.auto.commit", "true");
props.put("auto.commit.interval.ms", "1000");
props.put("fetch.message.max.bytes", 1024 * 1024);
props.put("session.timeout.ms", "30000");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("foo", "bar"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
        System.out.printf("offset = %d, key = %s, value = %s", record.offset(), record.key(), record.value());
}
```

Consumer state diagram



Consumer启动流程

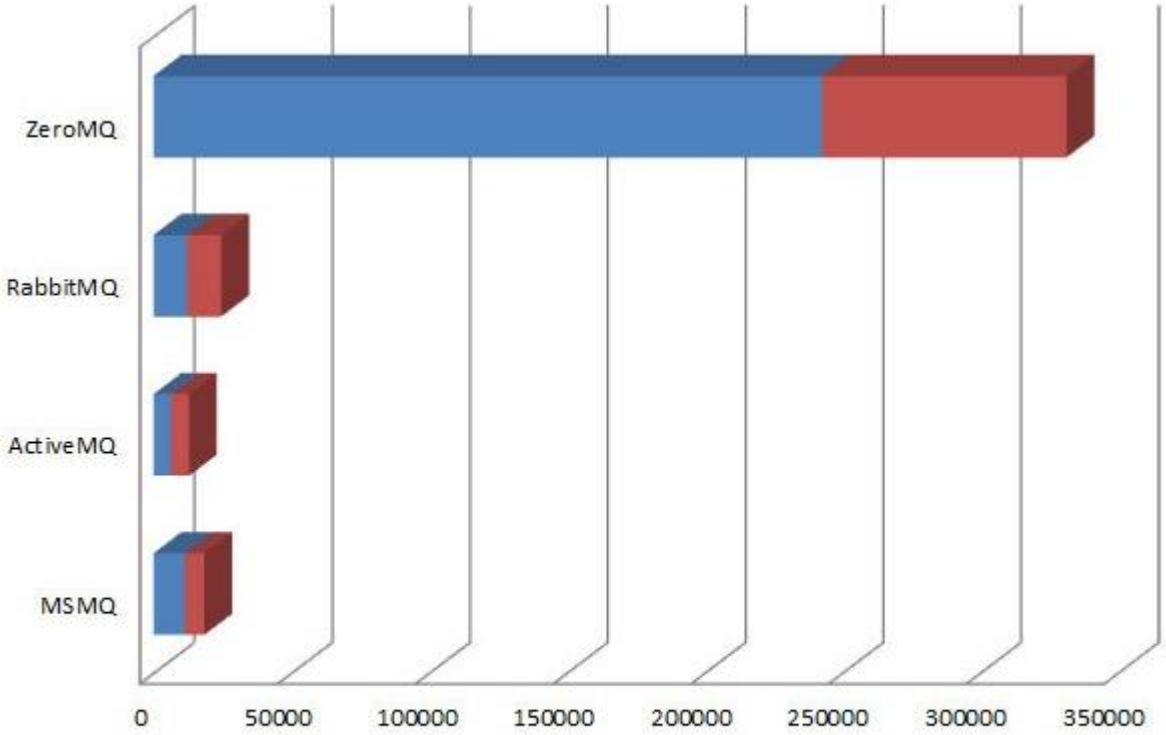


性能

- 其它MQ的吞吐量
- 消息大小 VS.吞吐量
- 副本数量 VS.吞吐量

其它MQ的吞吐量

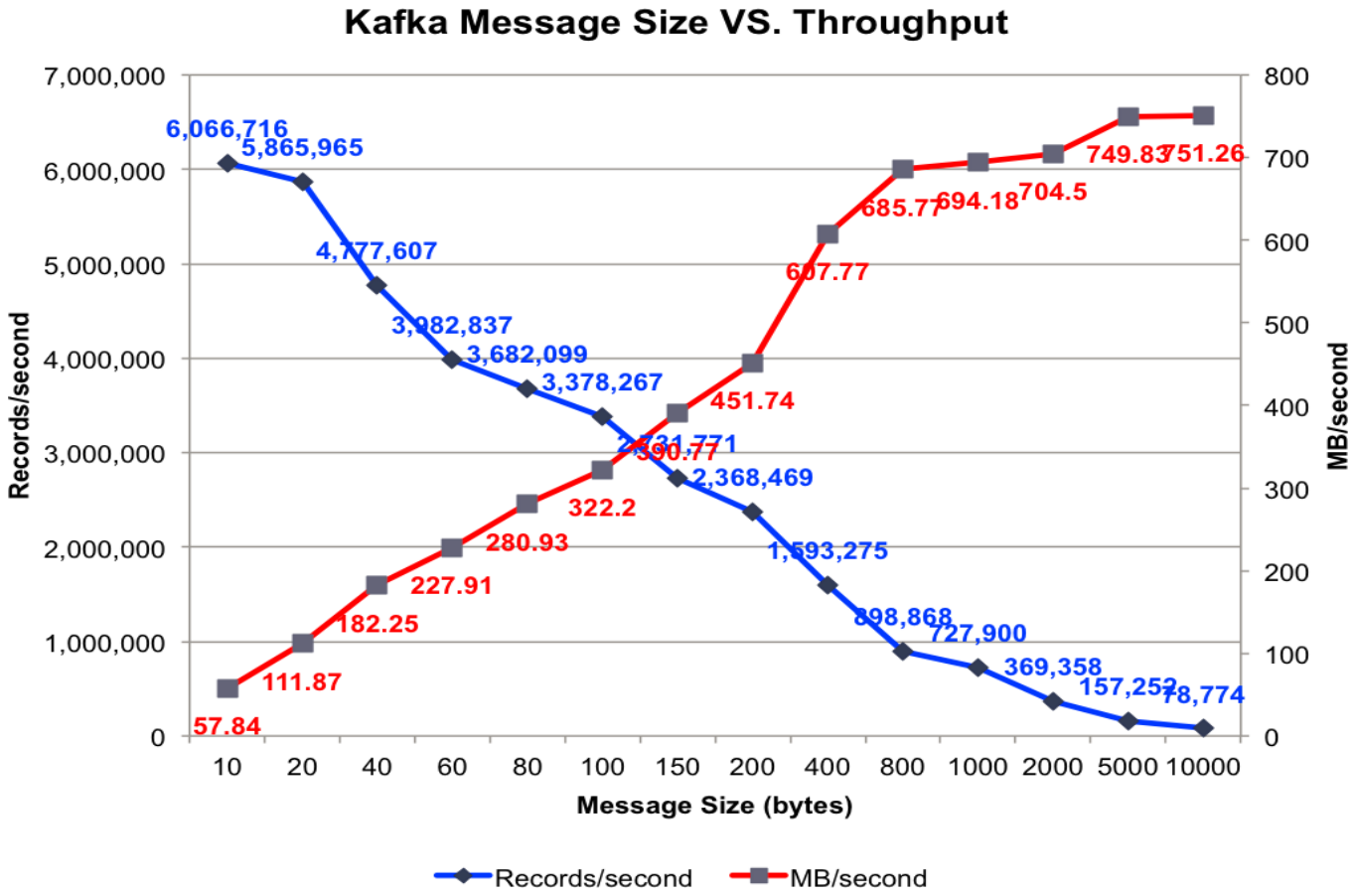
- 显示的是发送和接受的每秒钟的消息数。整个过程共产生1百万条1K的消息。测试的执行是在一个Windows Vista上进行的。



	MSMQ	ActiveMQ	RabbitMQ	ZeroMQ
Send	11406	6452	12278	242659
Receive	7021	6453	12281	88699

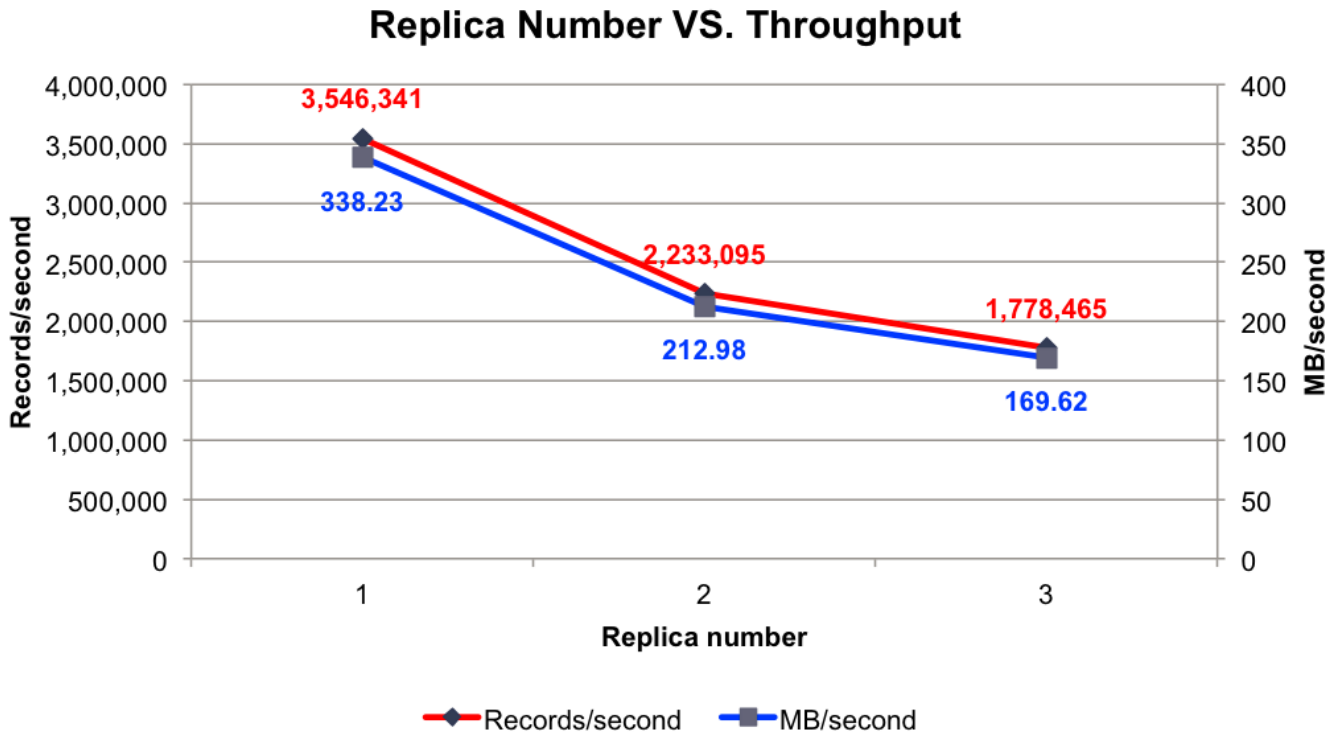
消息大小 VS.吞吐量

- 实验条件：3个Broker，1个Topic，6个Partition，无Replication，异步模式，3个Producer。
- 测试项目：分别测试消息长度为10，20，40，60，80，100，150，200，400，800，1000，2000，5000，10000字节时的集群总吞吐量。
- 测试结果：不同消息长度时的集群总吞吐率如下图所示：



副本数量 VS.吞吐量

- 实验条件：3个Broker，1个Topic，6个Partition，异步模式，3个Producer，消息Payload为100字节。
- 测试项目：分别测试1到3个Replica时的吞吐率。
- 测试结果：如下图所示：

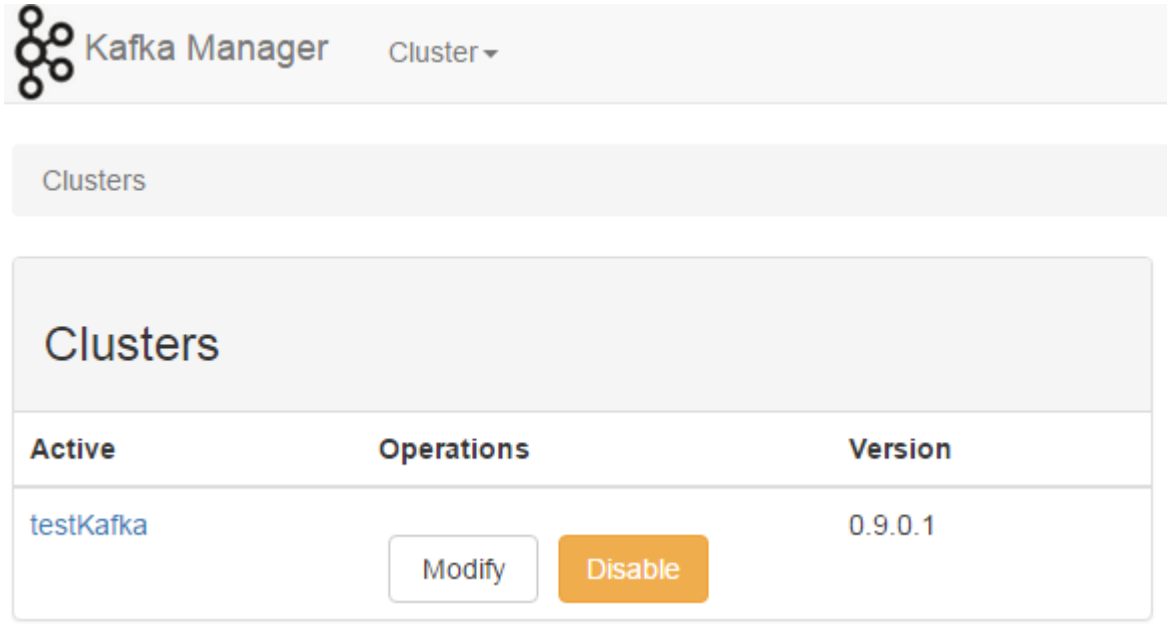


监控

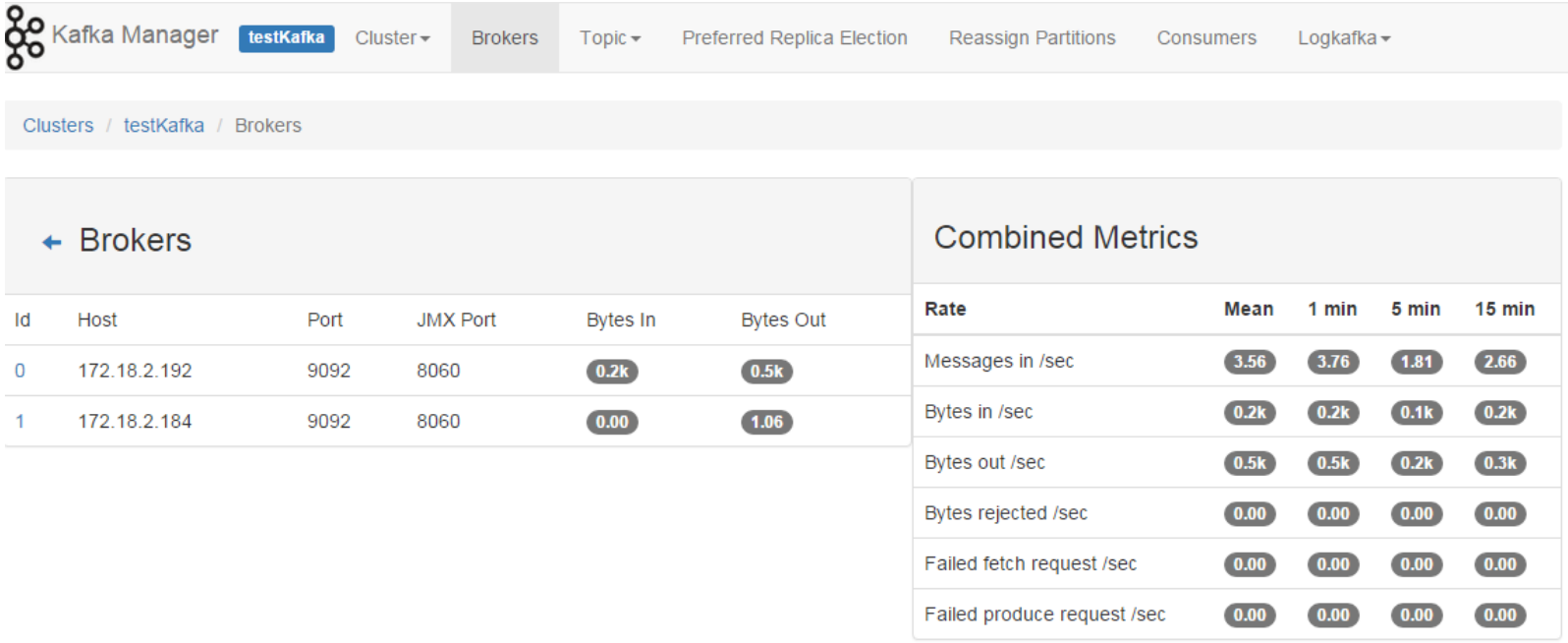
KafkaManager是Yahoo开源的一款Kafka集群监控管理工具，功能如下:

- 管理复数Kafka集群
- 查看Brokers信息
- 管理Topic(查看、创建、删除)
- 管理Partition(查看、添加、再分配)

Cluster


Kafka Manager
Cluster
Clusters
Clusters
Active
Operations
Version
testKafka
Modify
Disable
0.9.0.1

Brokers


Kafka Manager
testKafka
Cluster
Brokers
Topic
Preferred Replica Election
Reassign Partitions
Consumers
Logkafka
Clusters / testKafka / Brokers
Brokers
Id Host Port JMX Port Bytes In Bytes Out
0 172.18.2.192 9092 8060 0.2k 0.5k
1 172.18.2.184 9092 8060 0.00 1.06
Combined Metrics
Rate Mean 1 min 5 min 15 min
Messages in /sec 3.56 3.76 1.81 2.66
Bytes in /sec 0.2k 0.2k 0.1k 0.2k
Bytes out /sec 0.5k 0.5k 0.2k 0.3k
Bytes rejected /sec 0.00 0.00 0.00 0.00
Failed fetch request /sec 0.00 0.00 0.00 0.00
Failed produce request /sec 0.00 0.00 0.00 0.00