Building Pipeline/Streaming Application with Apache Kafka

zqhxuyuan.github.io
2017.10

Apache Kafka：从消息中间件到流处理平台

Apache Kafka

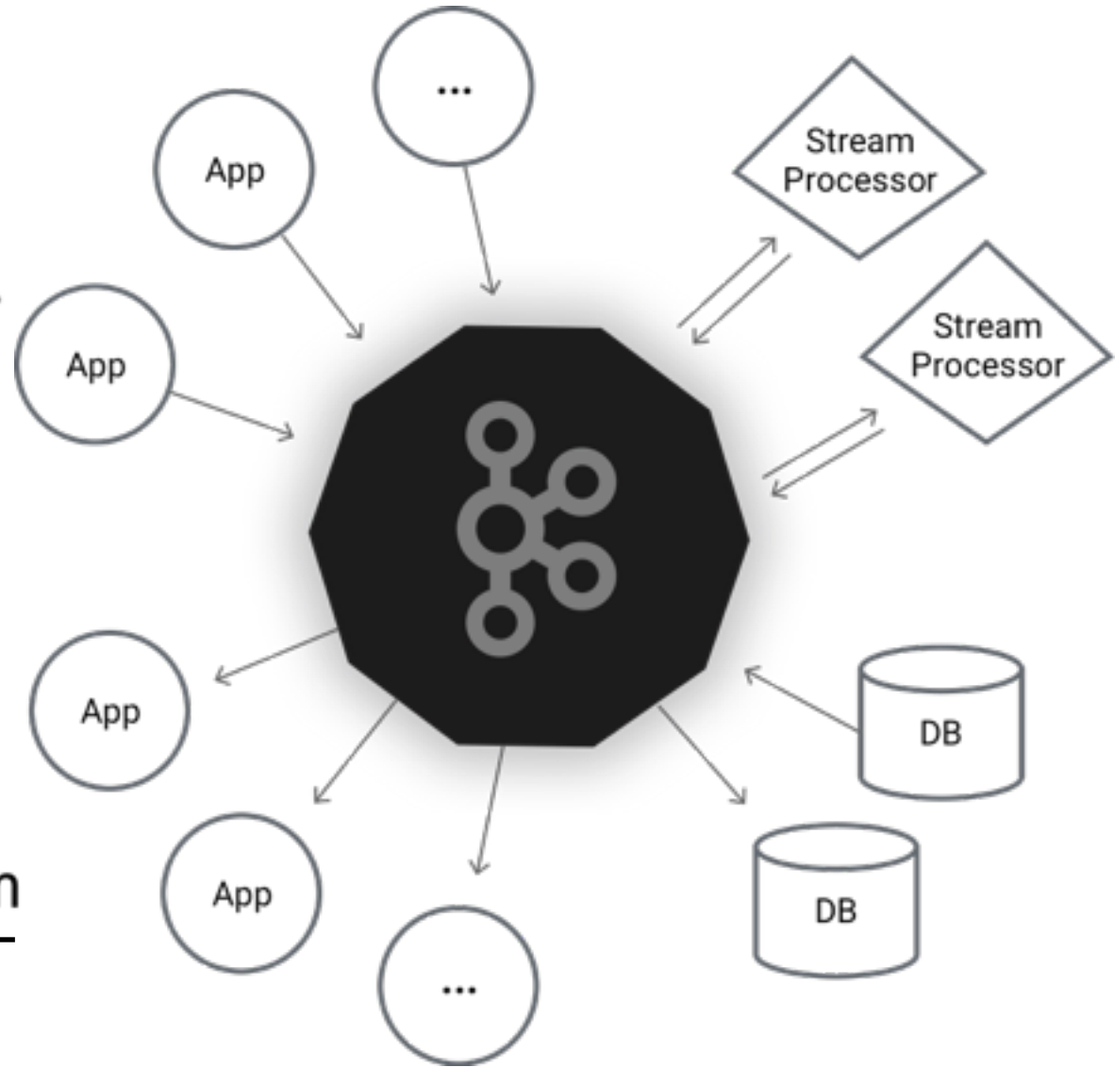
A high-throughput distributed messaging system.

消息系统

APACHE kafka™

A distributed streaming platform

流处理平台



TURING 图灵原创

- 图文详解Kafka的内部原理、设计与实现
- 全面分析以Kafka为中心的分布式流平台
- Kafka新特性详解，包括连接器和流处理

Kafka技术内幕

图文详解Kafka源码设计与实现

郑奇煌◎著

起源于LinkedIn的Kafka自2012年进入Apache软件基金会以来，其产品和社区得到了蓬勃发展，它在云计算和大数据技术栈中也扮演着重要角色。最近两年，Kafka将重心逐步转向Streaming，通过内置流计算算子的形式提供一种轻量级的流计算解决方案。本书主要以Kafka 0.10.x版本的源码为基础，细致全面地剖析了Kafka各个组件的实现原理。如果你希望了解Kafka背后的一些设计理念，强烈推荐此书！

——冯嘉(Von Gosling)

阿里巴巴高级技术专家，Aliware MQ 总架构师

Apache RocketMQ联合创始人，Linux OpenMessaging规范发起人

Kafka是当前大数据解决方案的标配，广泛用于大数据框架间的数据发布和订阅，所以深入理解Kafka内部机制就非常必要。本书从快速入门到深入源码剖析，详细解构了Kafka内部各个部分，特别是存储、发布订阅、协调控制、流处理等。本书的内容不错，值得推荐！

——时金魁

华为云主任工程师

最近几年Kafka在国内外发展势头非常不错，已经成为一个既定的事实标准，但市面上介绍Kafka的图书非常少，而这本书以图文并茂的形式全面介绍了Kafka各个模块的运行原理。本书全面深入地介绍了Kafka，是一本不可多得的技术书。

——吴阳平

过往记忆技术博客博主，Qunar数据架构师

中国工信出版集团　人民邮电出版社 POSTS & TELECOM PRESS

图灵社区：iTuring.cn

微　　博：@图灵教育　@图灵社区

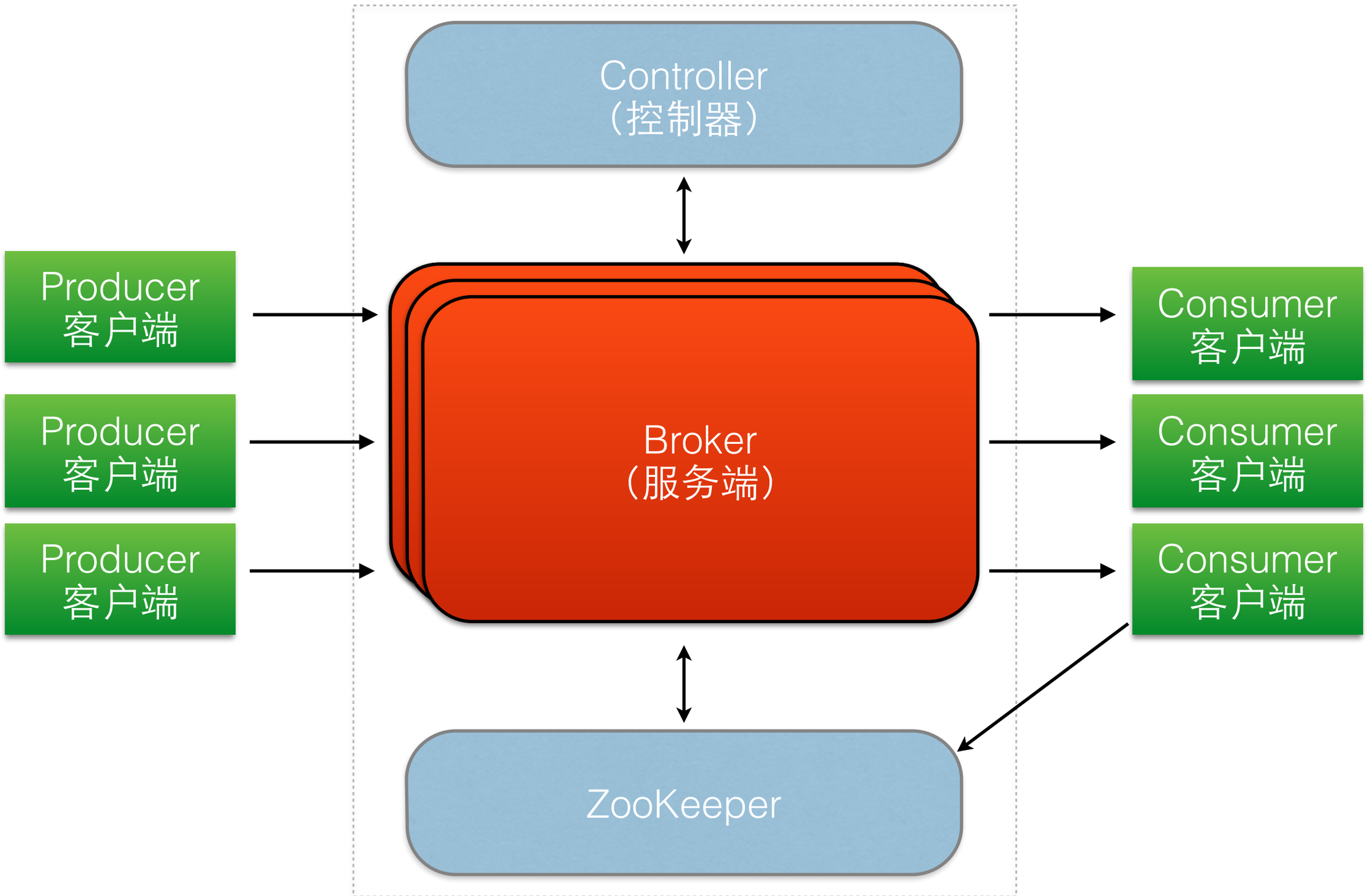
分类建议 计算机／大数据／Kafka

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-46938-0

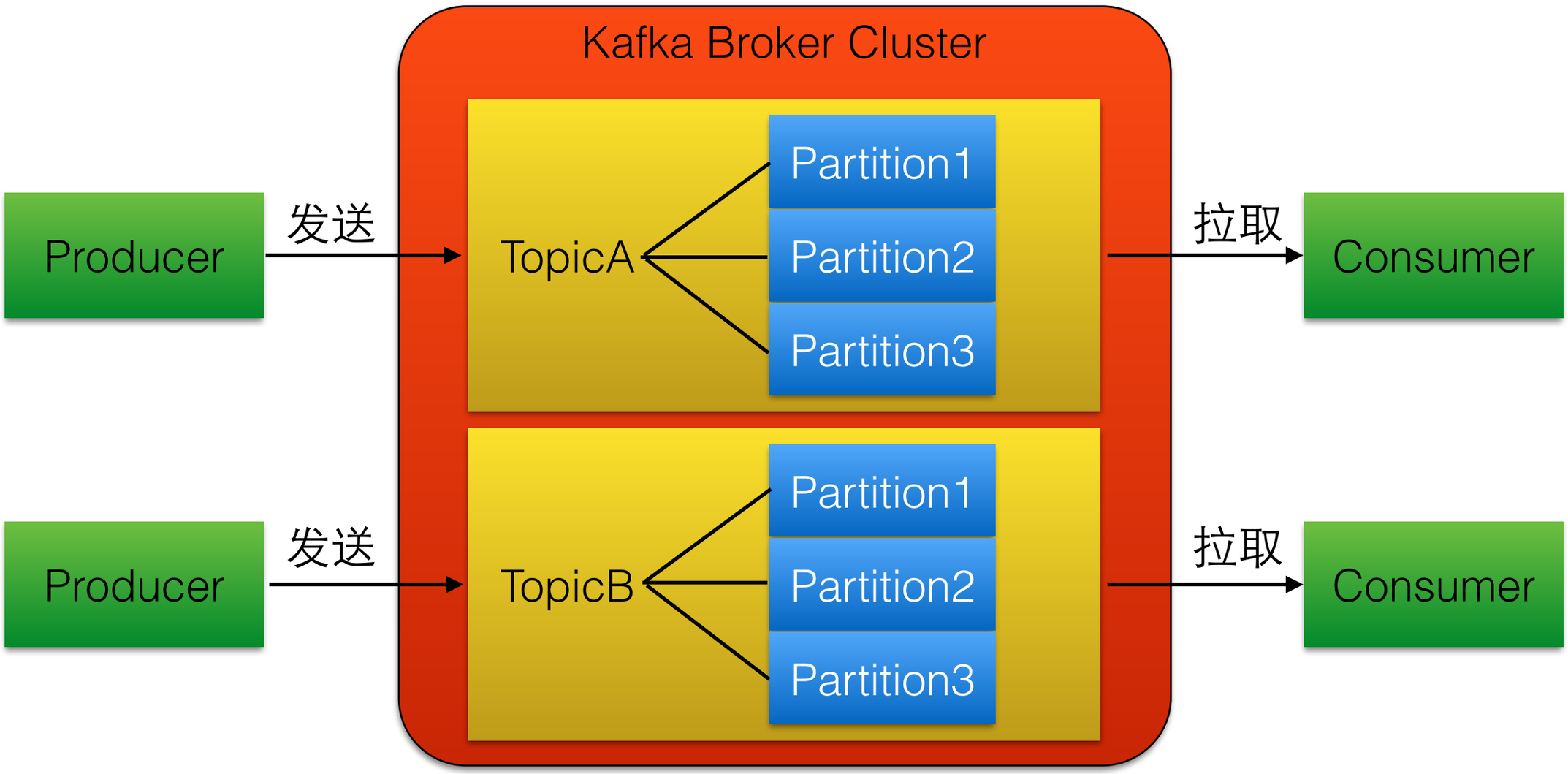
定价：119.00元

Kafka Architecture & Components

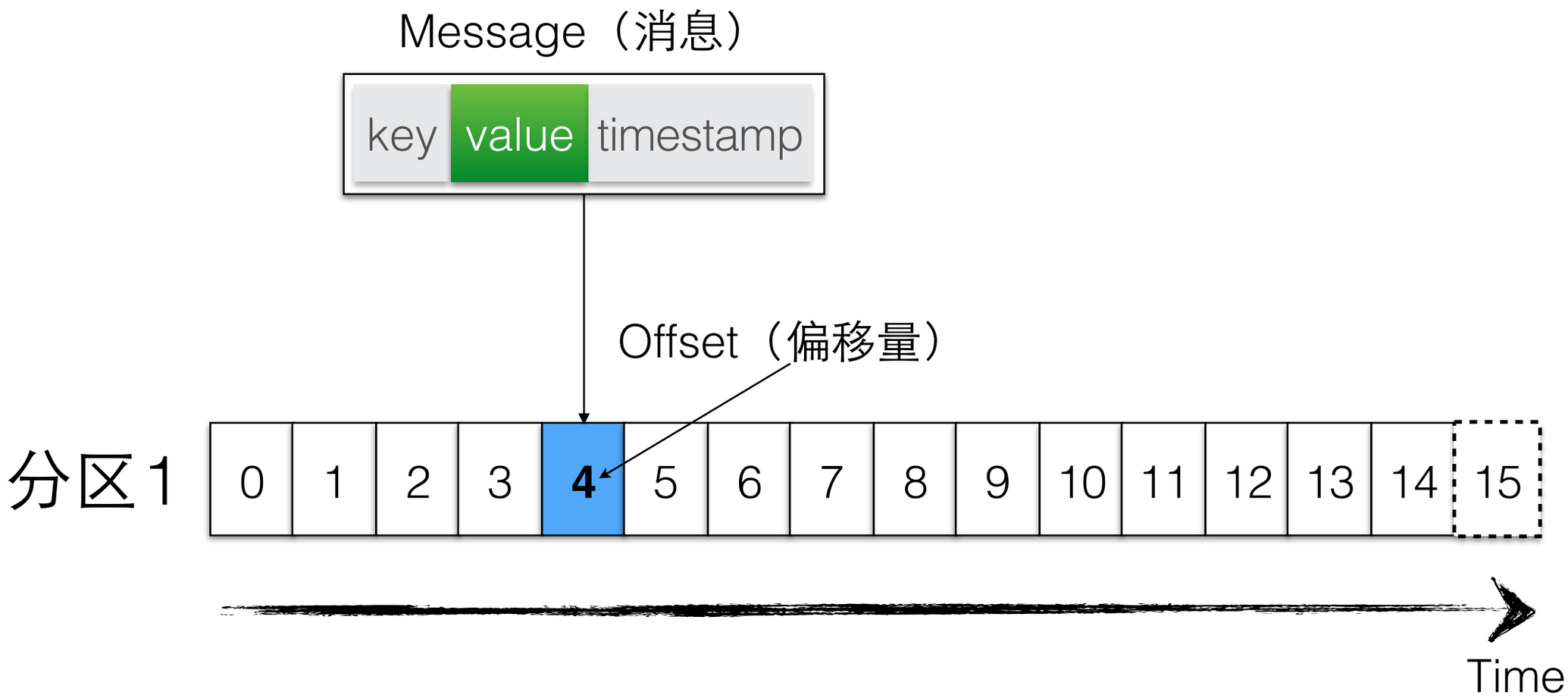


基本概念
- 主题（Topic）
- 分区（Partition）

对比	Kafka	分布式数据库中间件
逻辑 概念	Topic （主题）	“Table” （MyCat表）
物理 存储	Partition （分区）	Sharding （分库分表）

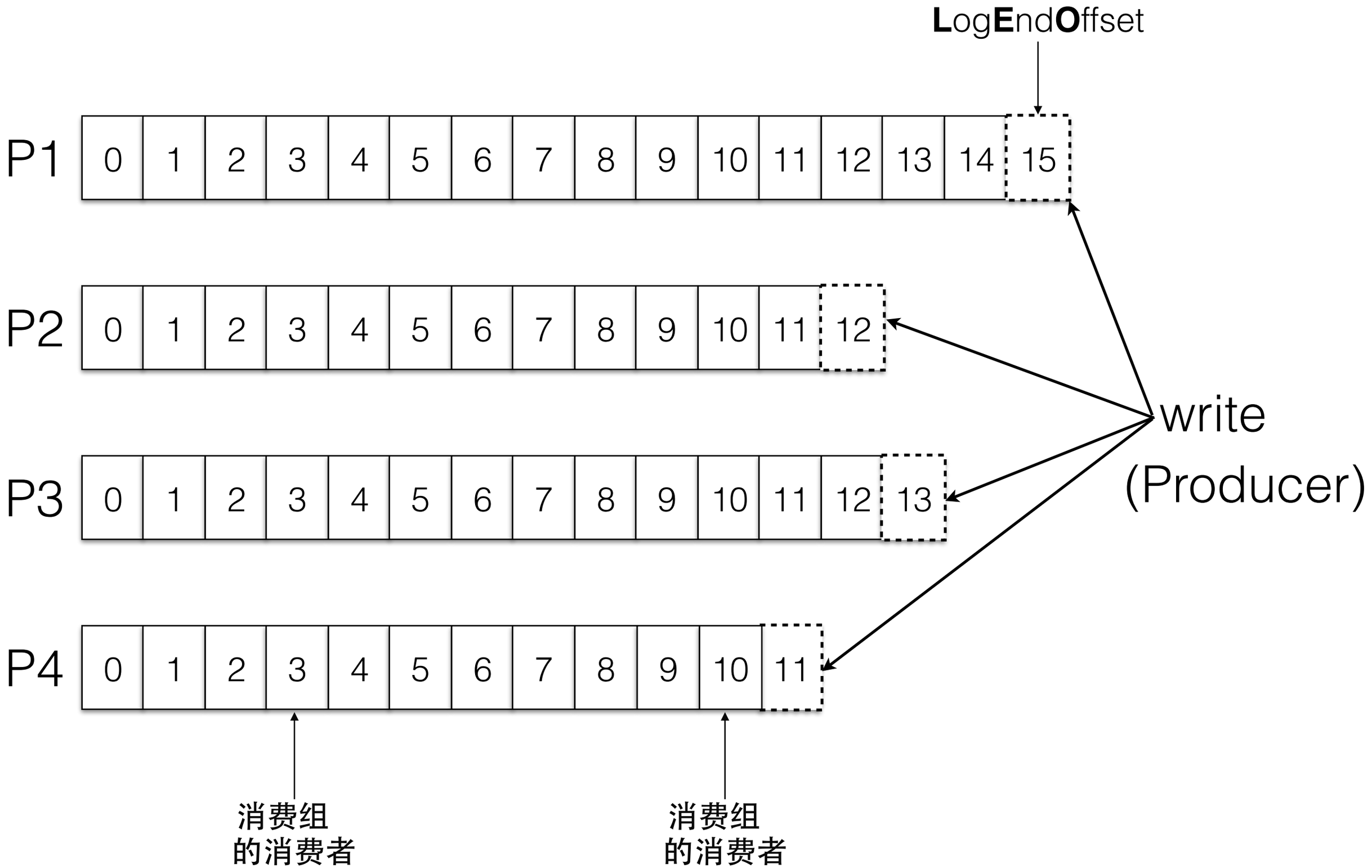


Partition



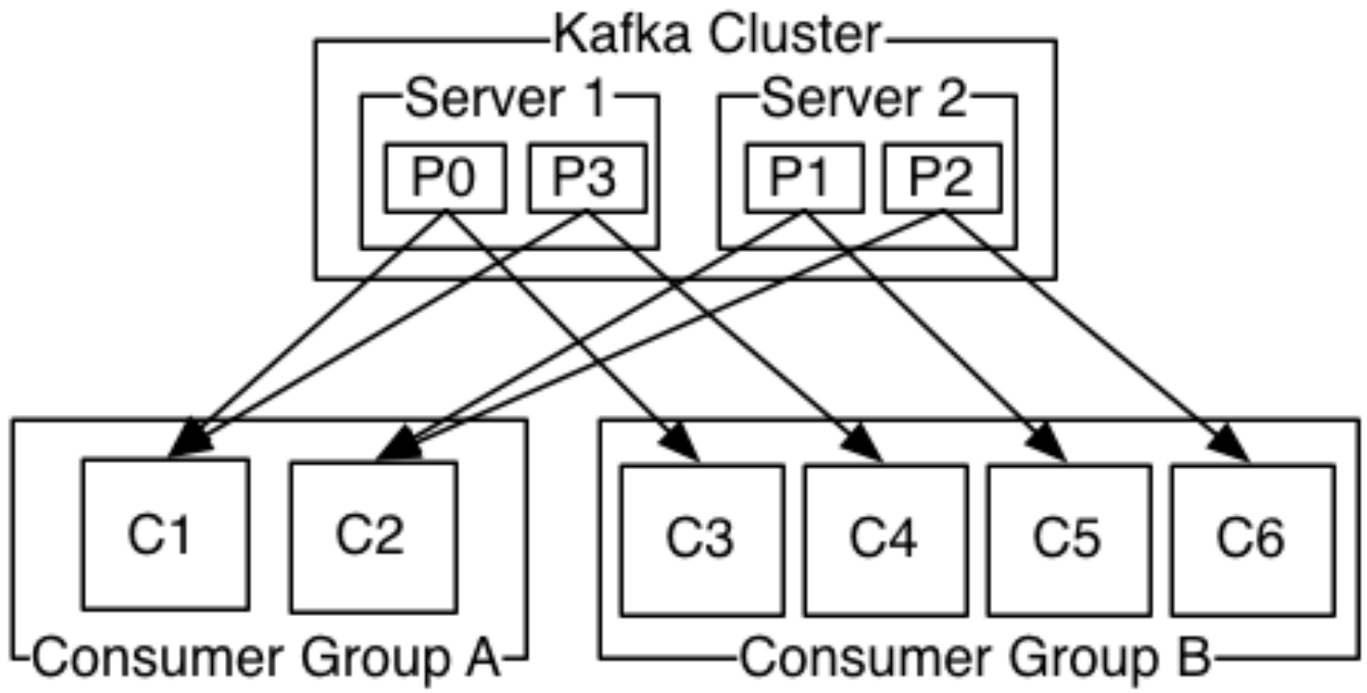
消息的偏移量永远单调递增

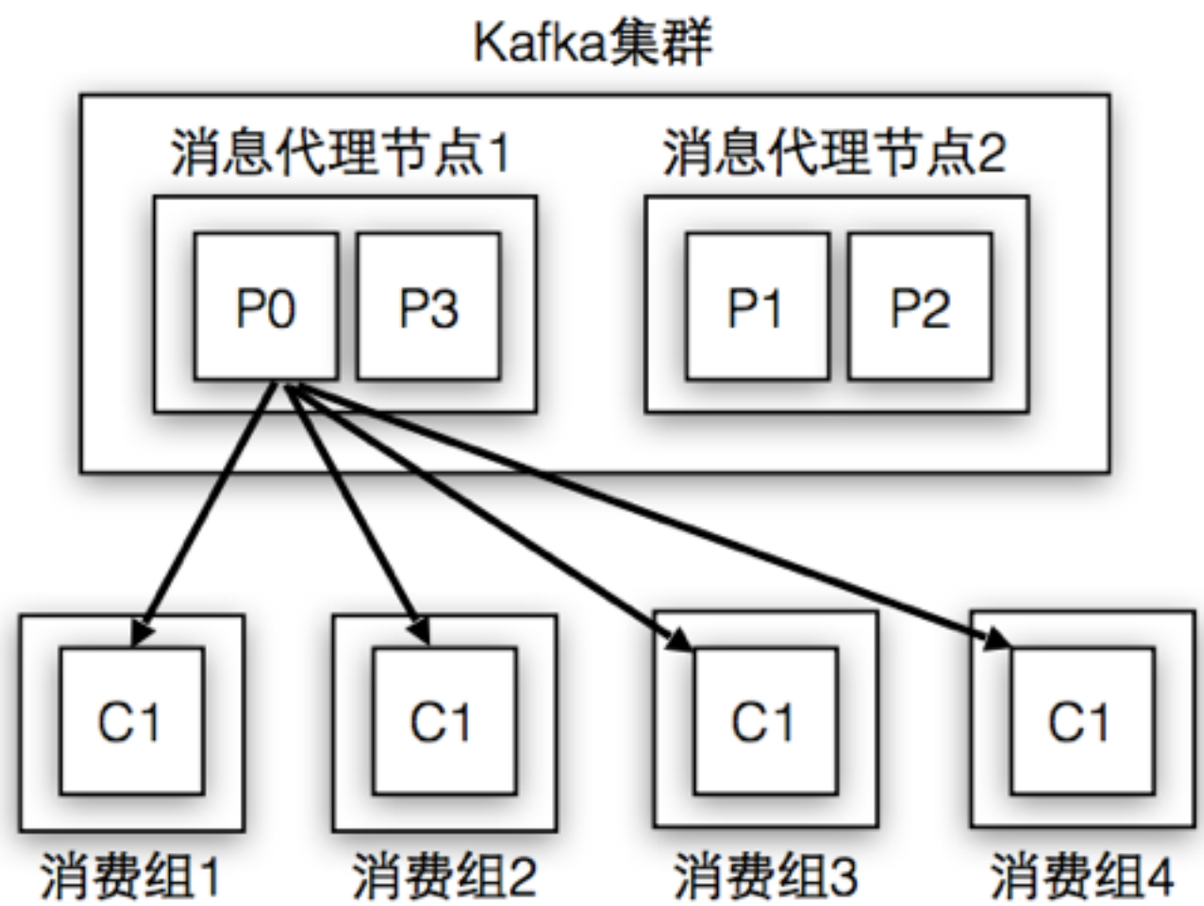
Topic


LogEndOffset
P1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
P2
0 1 2 3 4 5 6 7 8 9 10 11 12
P3
0 1 2 3 4 5 6 7 8 9 10 11 12 13
P4
0 1 2 3 4 5 6 7 8 9 10 11
write
(Producer)
消费组
的消费者
消费组
的消费者

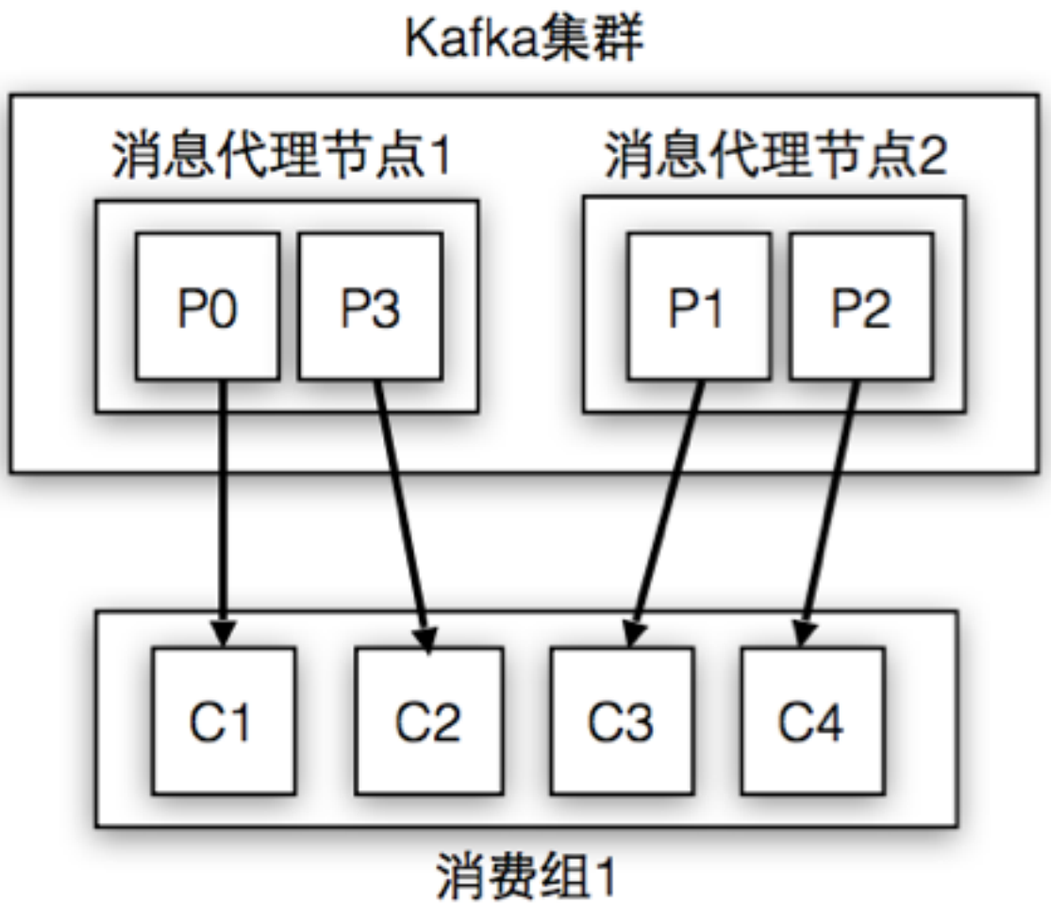
消息模型

- 广播
- 队列



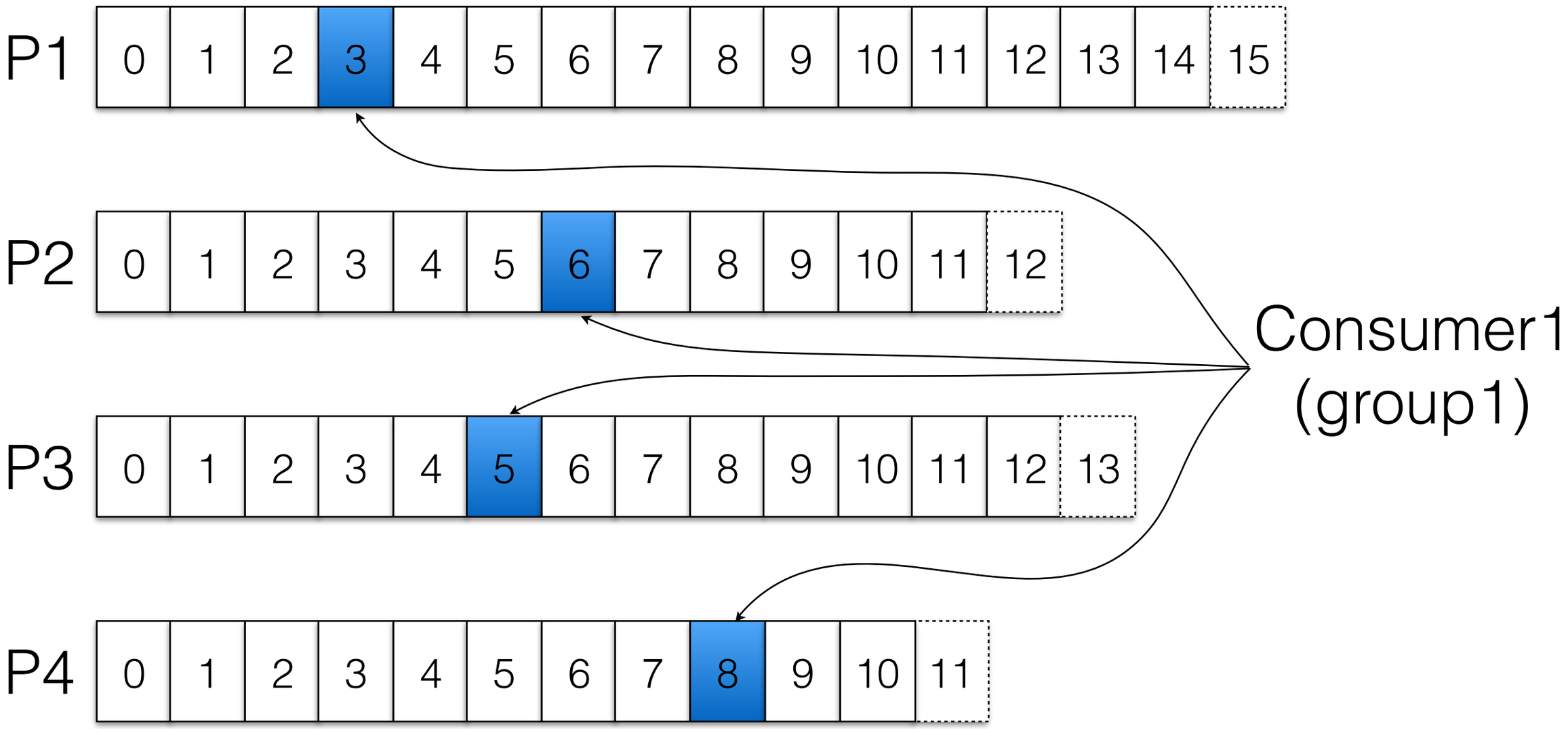


广播（发布-订阅模式）

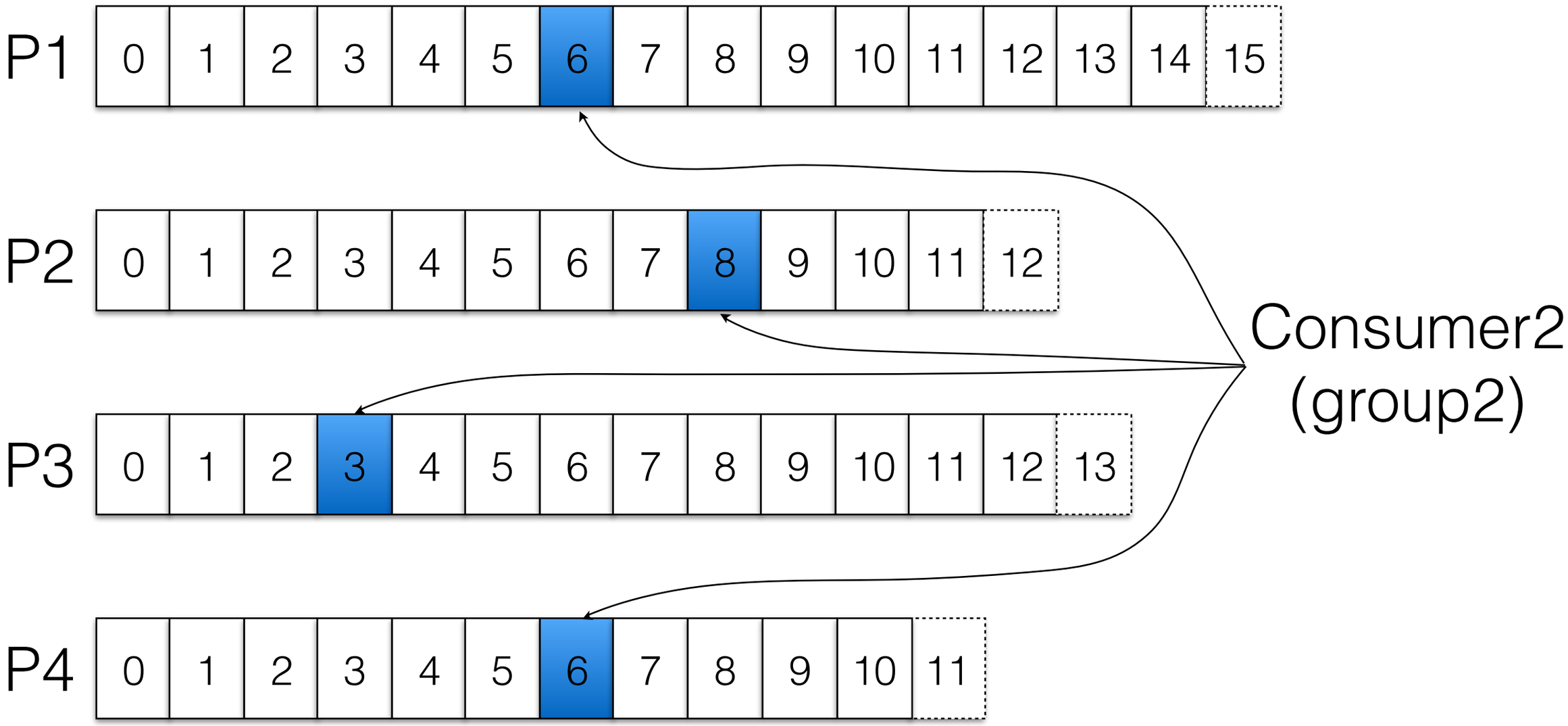


单播（队列模式）

Consumer Group1

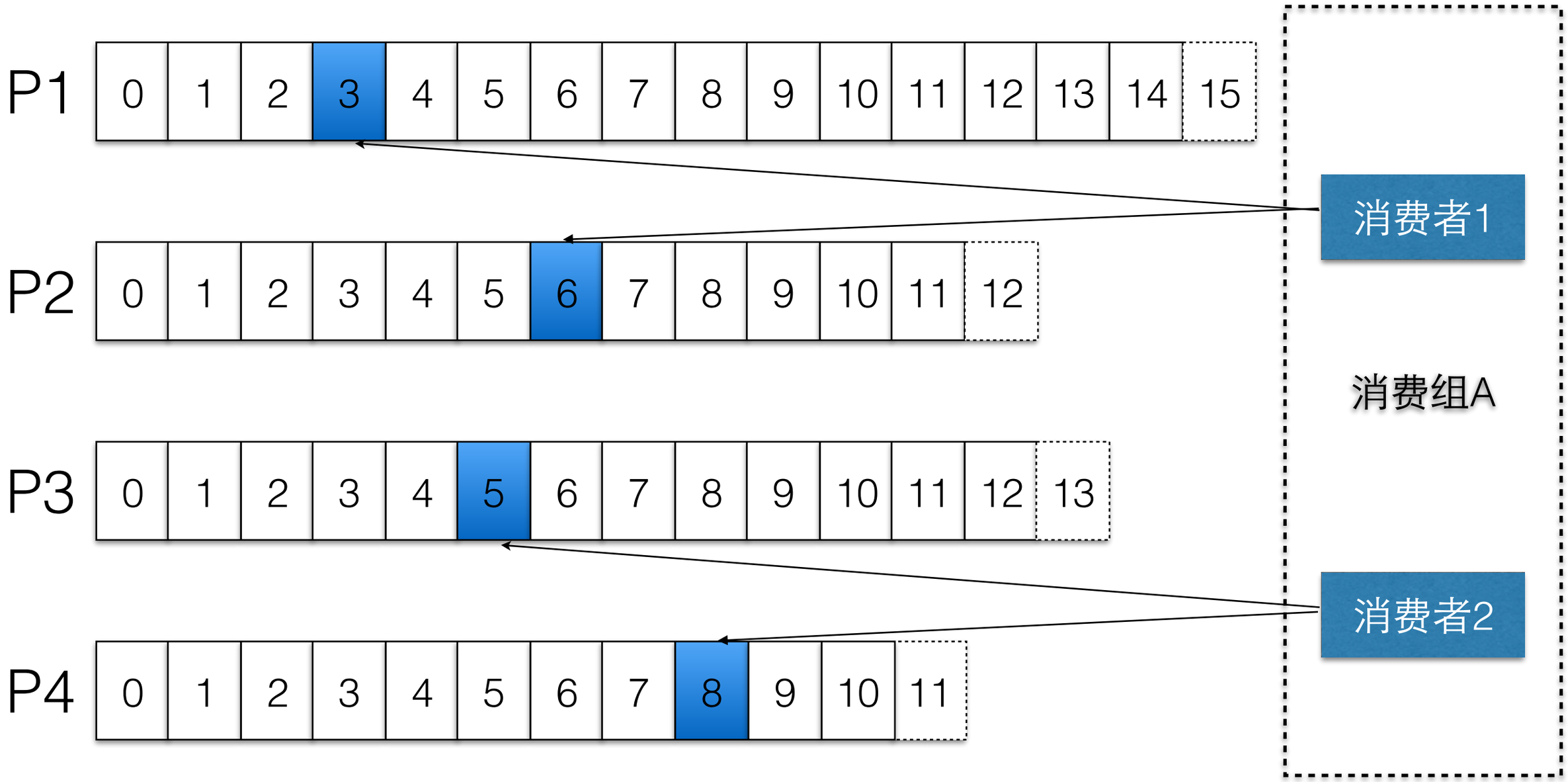


Consumer Group2



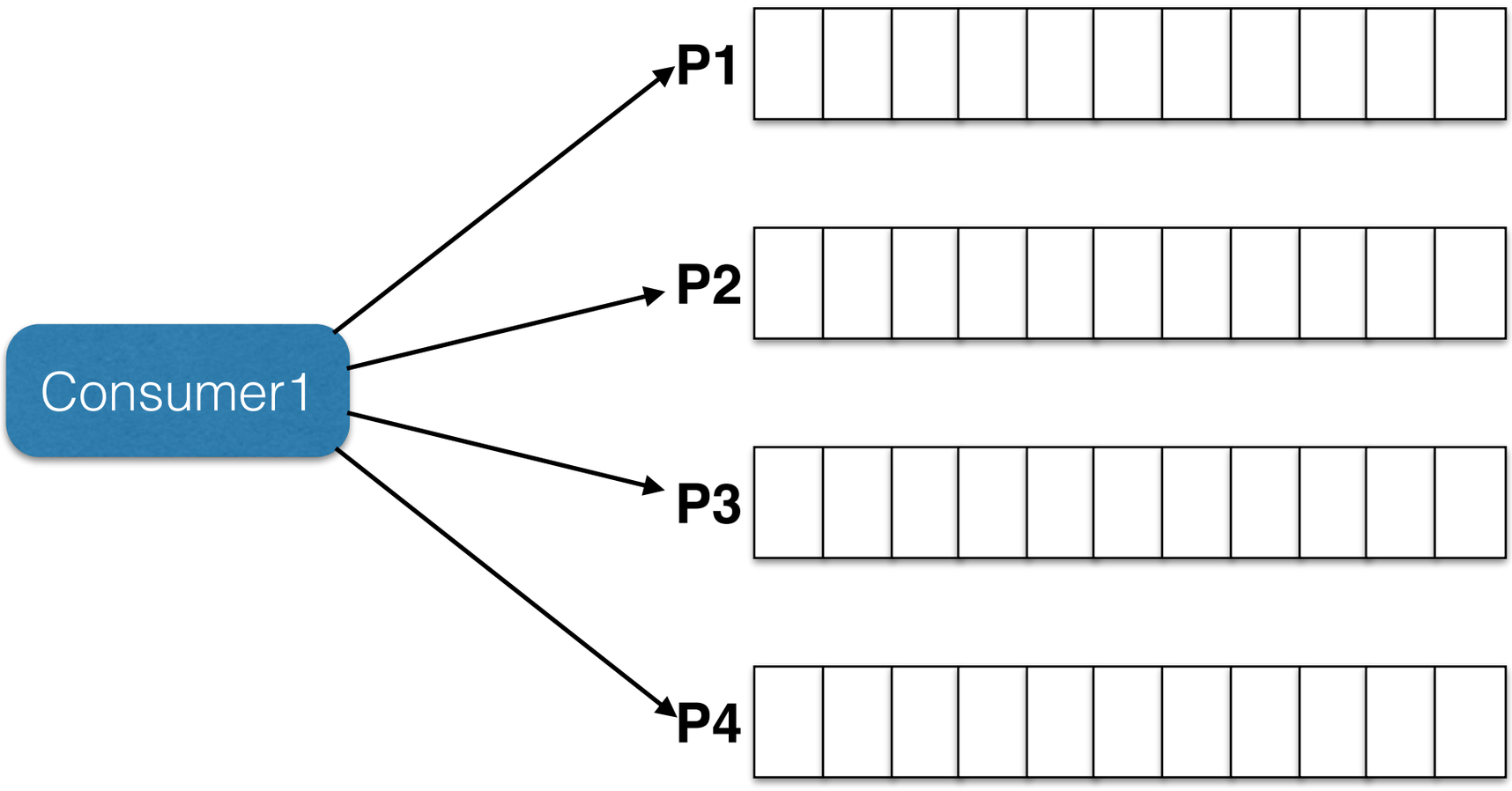
不同逻辑的消费组可以拉取相同的主题，互不影响

Consumers



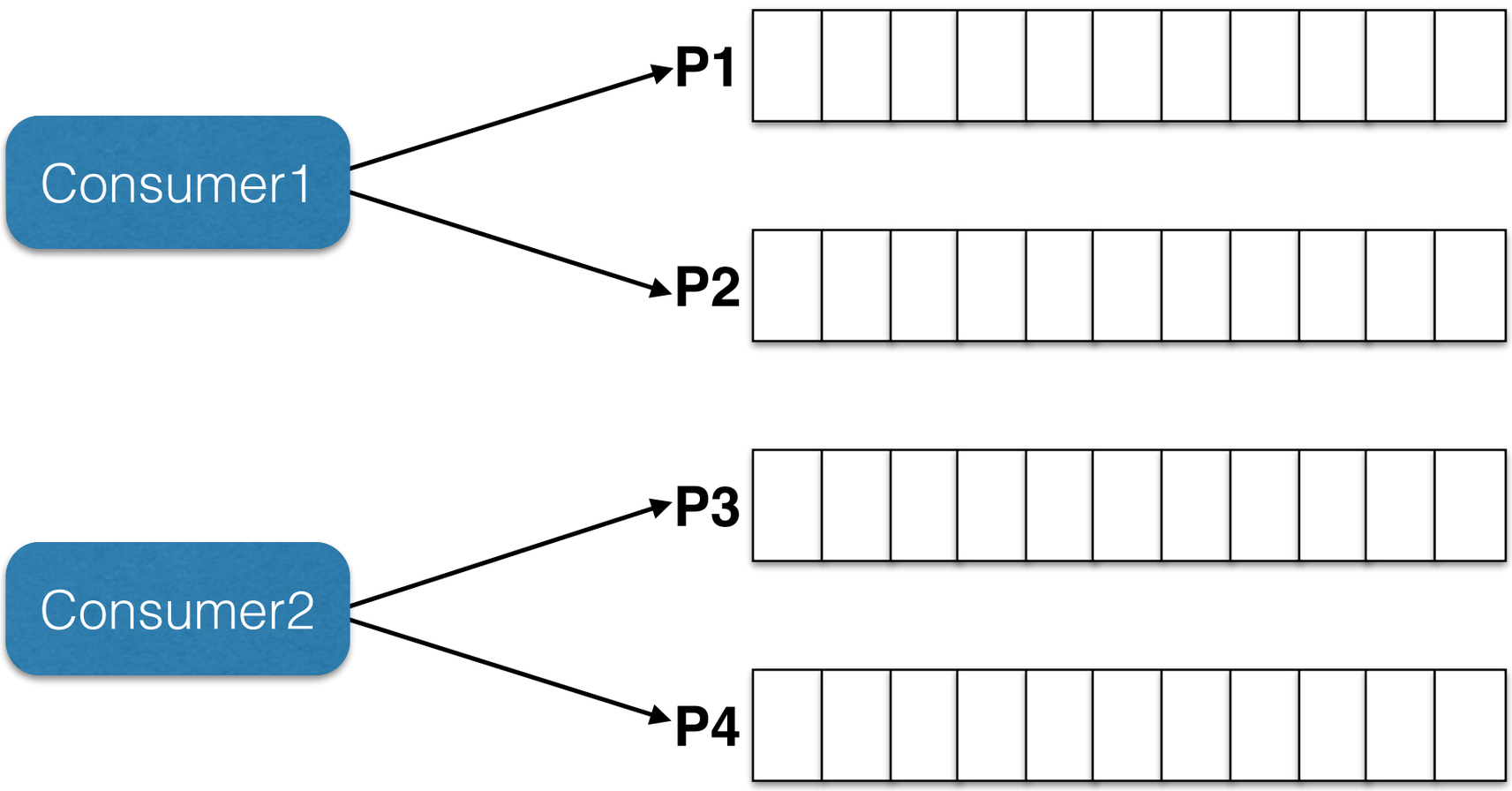
一个逻辑消费组可以包括多个消费者

1 Consumer

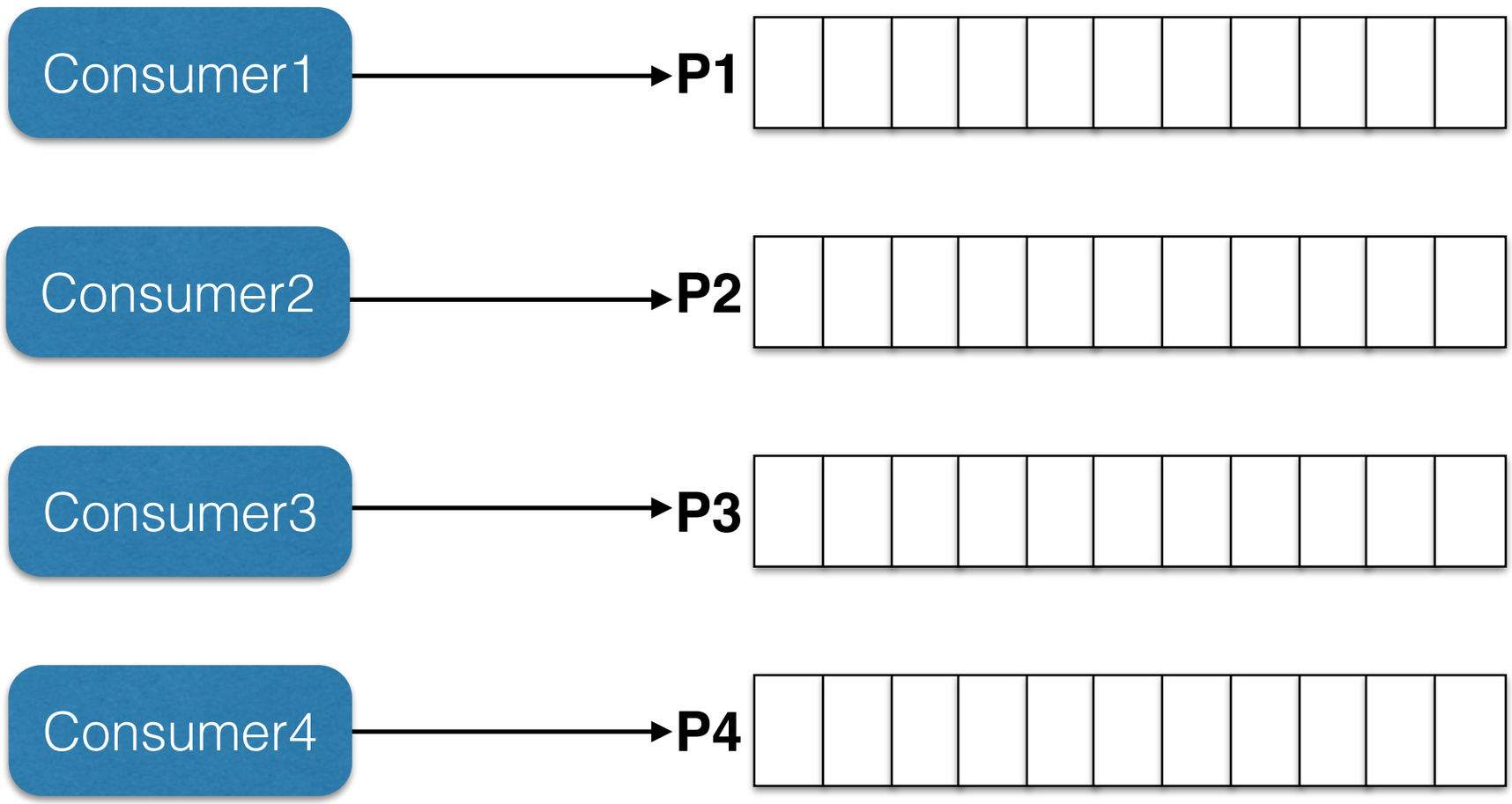


消费者分配分区的算法：RoundRobbin/Range

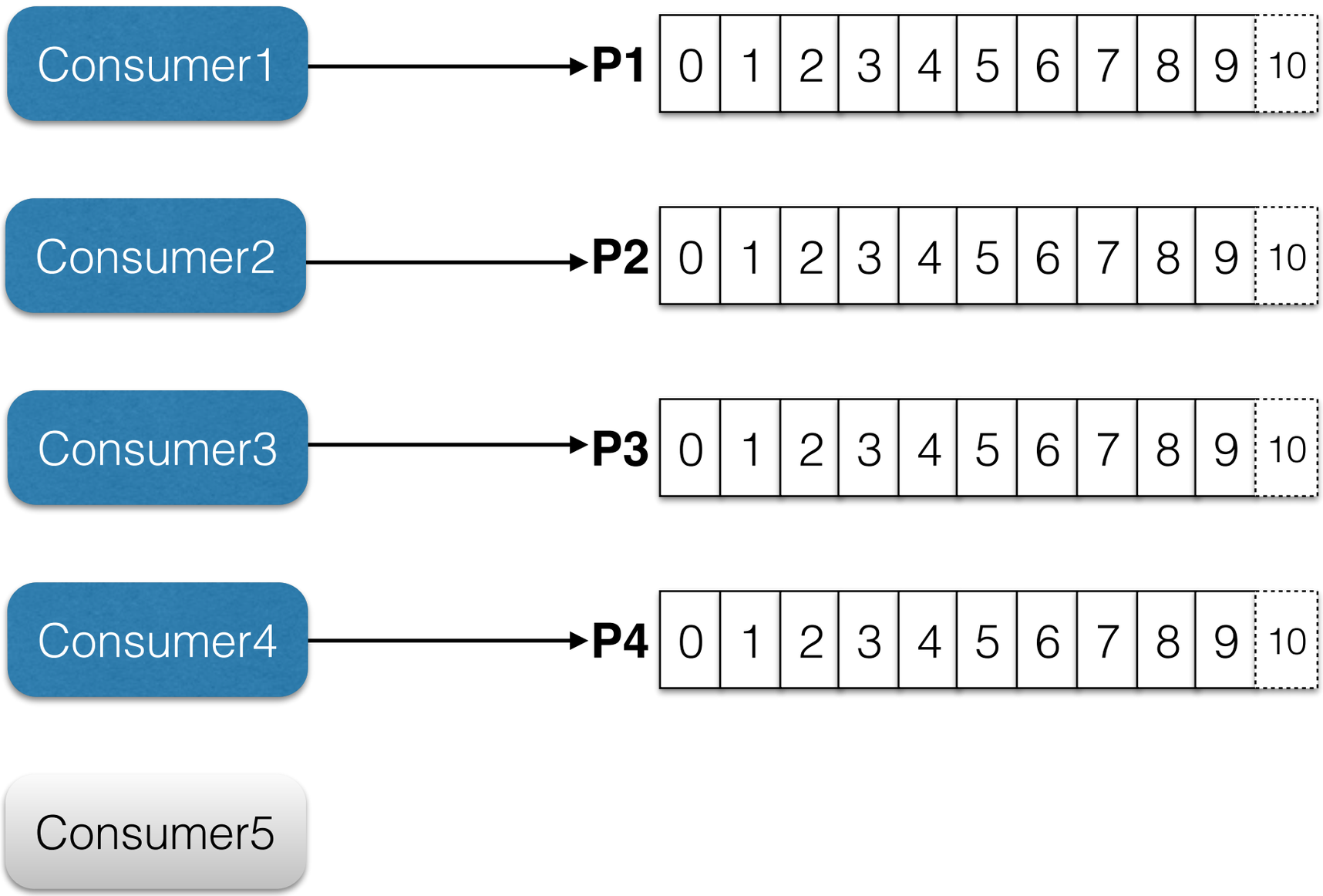
2 Consumers



4 Consumers

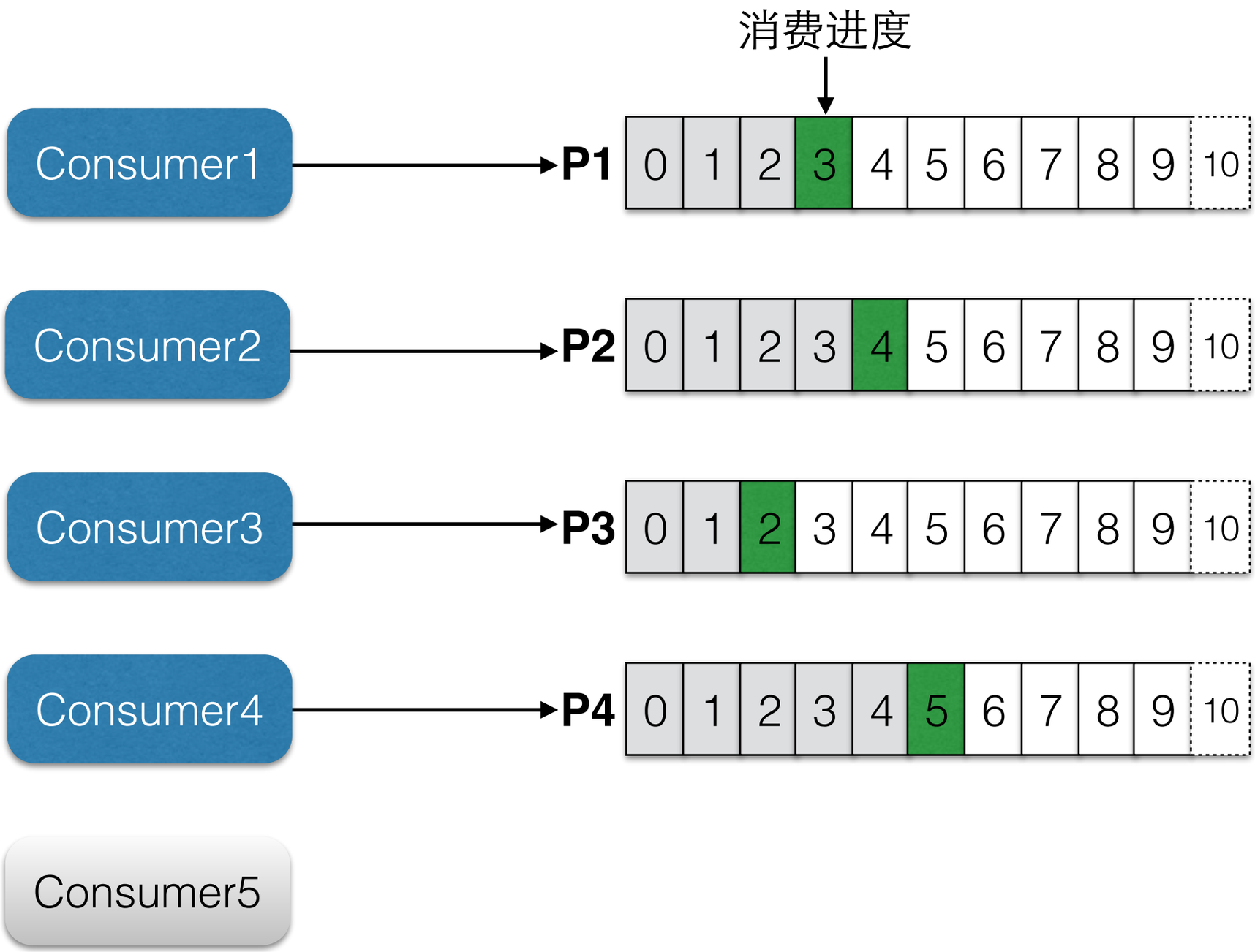


5 Consumers

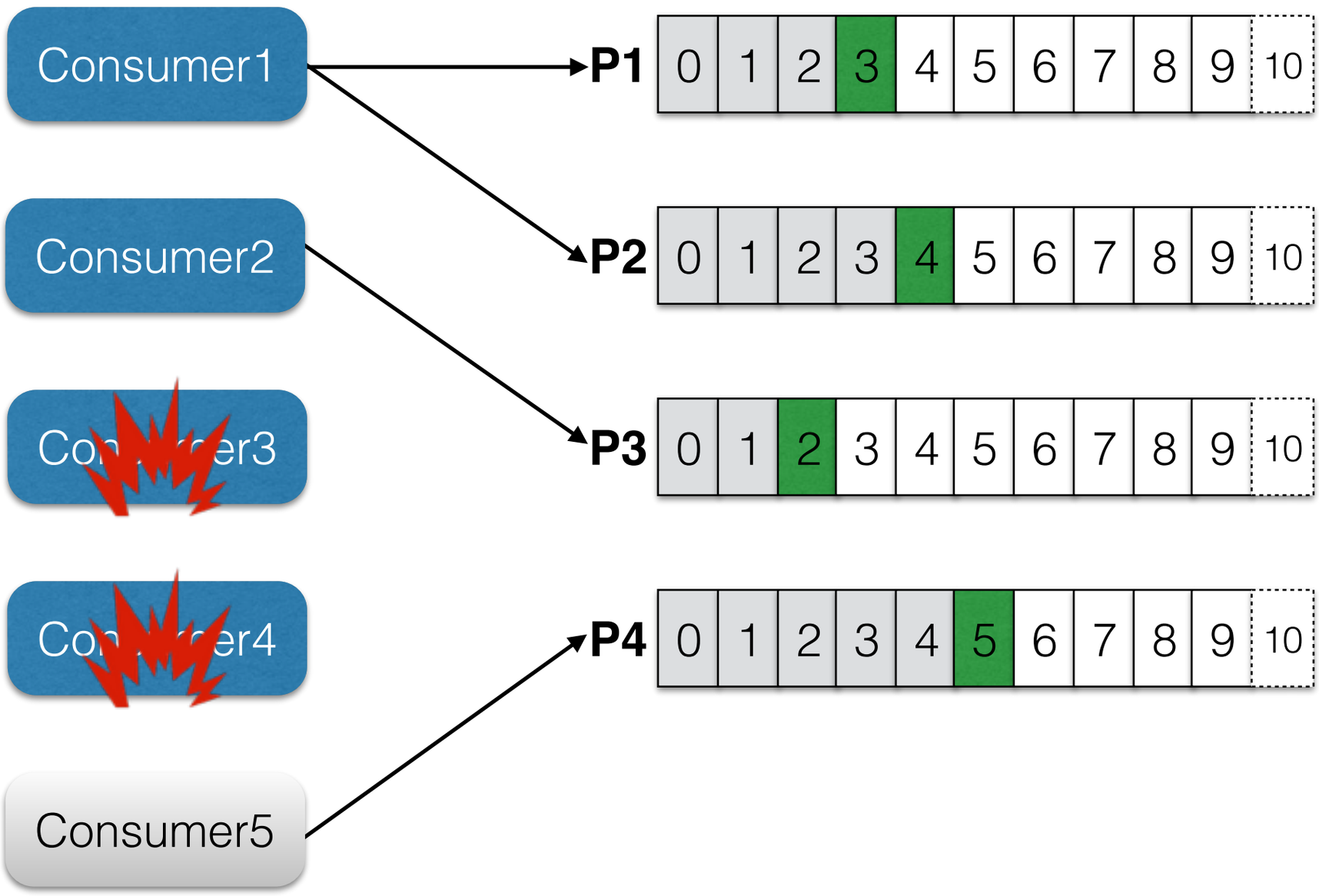


为什么一个分区只能被一个消费者所消费?
消息顺序问题、并发读写问题

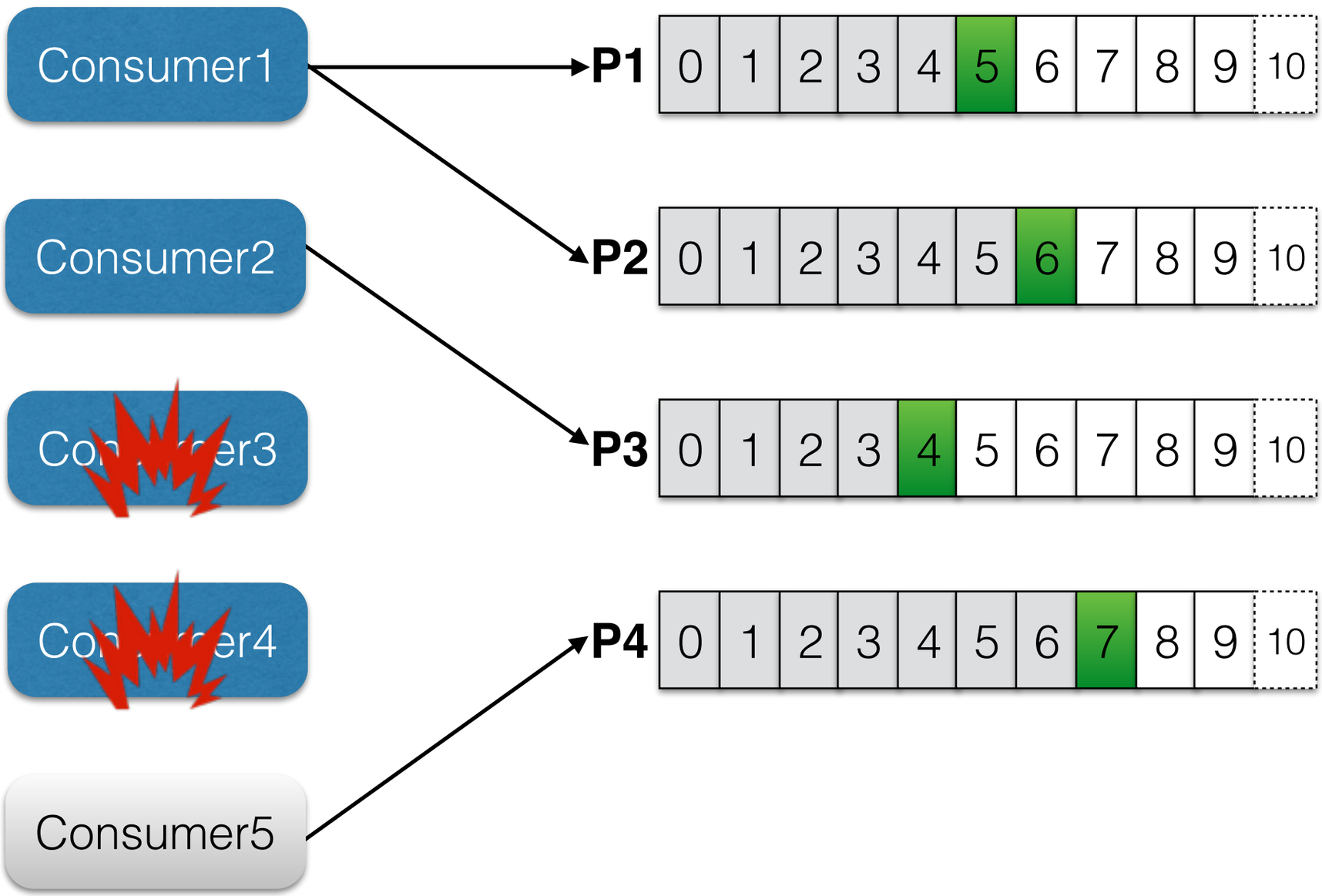
5 Consumers


消费进度
Consumer1
P1
0 1 2 3 4 5 6 7 8 9 10
Consumer2
P2
0 1 2 3 4 5 6 7 8 9 10
Consumer3
P3
0 1 2 3 4 5 6 7 8 9 10
Consumer4
P4
0 1 2 3 4 5 6 7 8 9 10
Consumer5

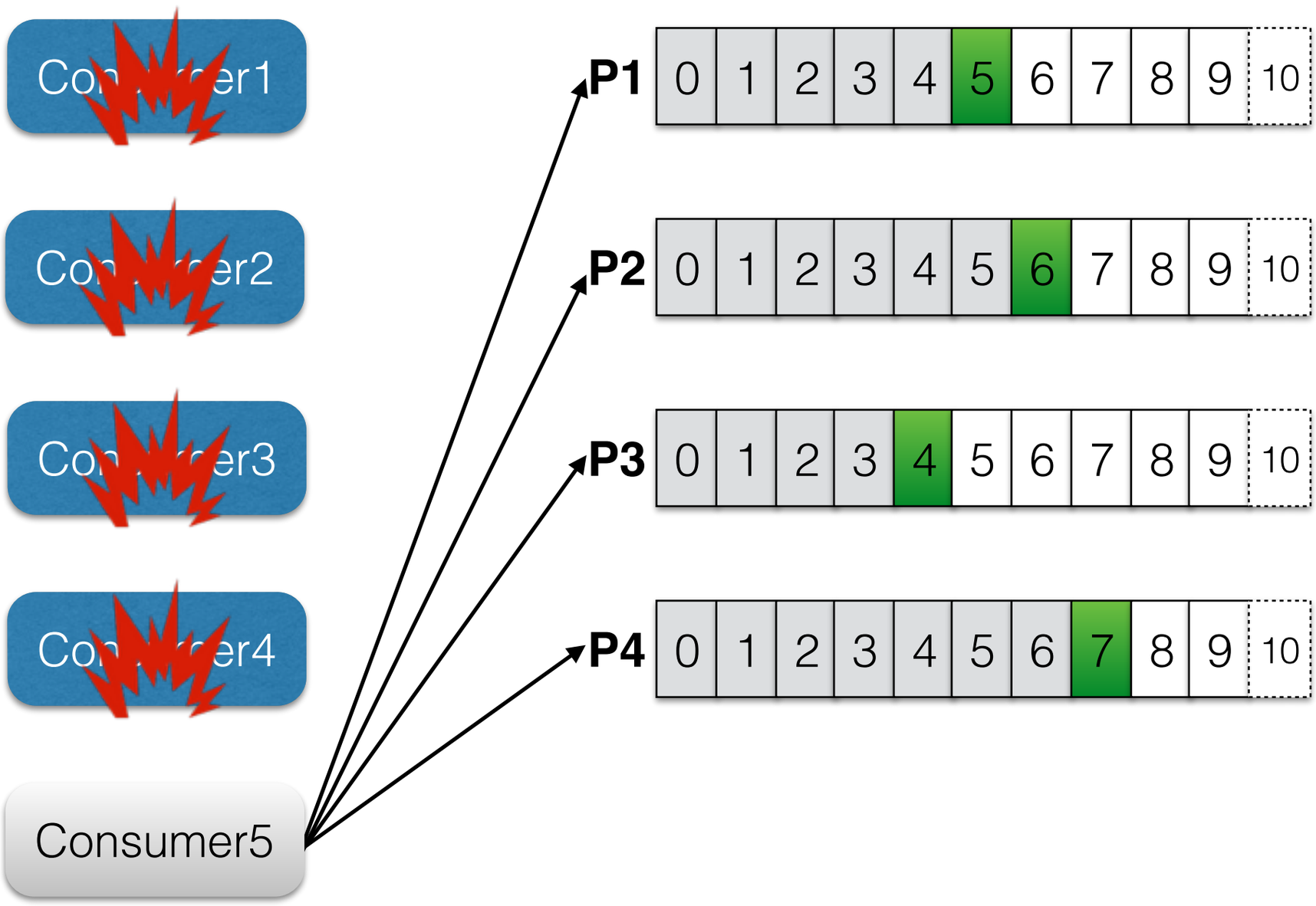
3 Consumers


Consumer1
Consumer2
Consumer3
Consumer4
Consumer5
P1
P2
P3
P4
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9 10

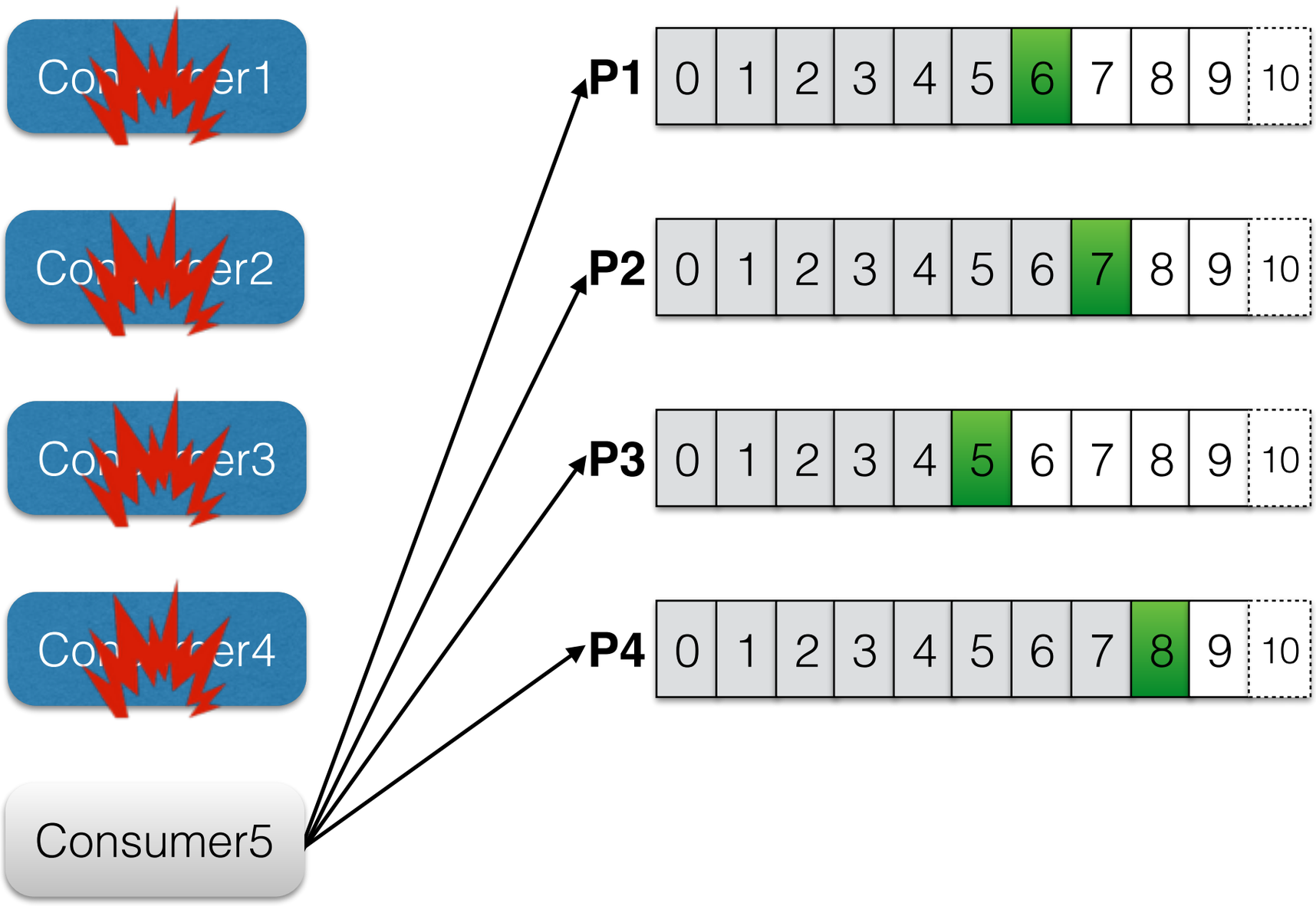
3 Consumers


Consumer1
Consumer2
Consumer3
Consumer4
Consumer5
P1
P2
P3
P4
0 1 2 3 4 5 6 7 8 9 10

1 Consumers

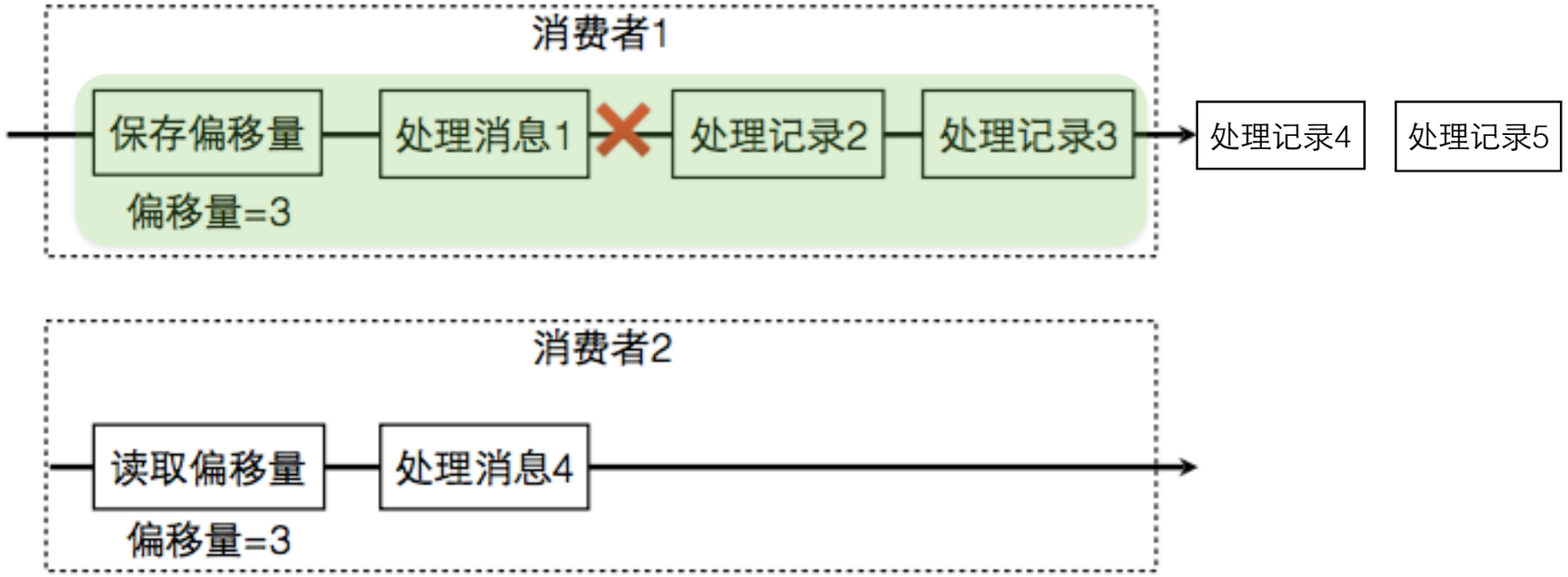

Consumer1
Consumer2
Consumer3
Consumer4
Consumer5
P1
0 1 2 3 4 5 6 7 8 9 10
P2
0 1 2 3 4 5 6 7 8 9 10
P3
0 1 2 3 4 5 6 7 8 9 10
P4
0 1 2 3 4 5 6 7 8 9 10

1 Consumers


Consumer1
Consumer2
Consumer3
Consumer4
Consumer5
P1
0 1 2 3 4 5 6 7 8 9 10
P2
0 1 2 3 4 5 6 7 8 9 10
P3
0 1 2 3 4 5 6 7 8 9 10
P4
0 1 2 3 4 5 6 7 8 9 10

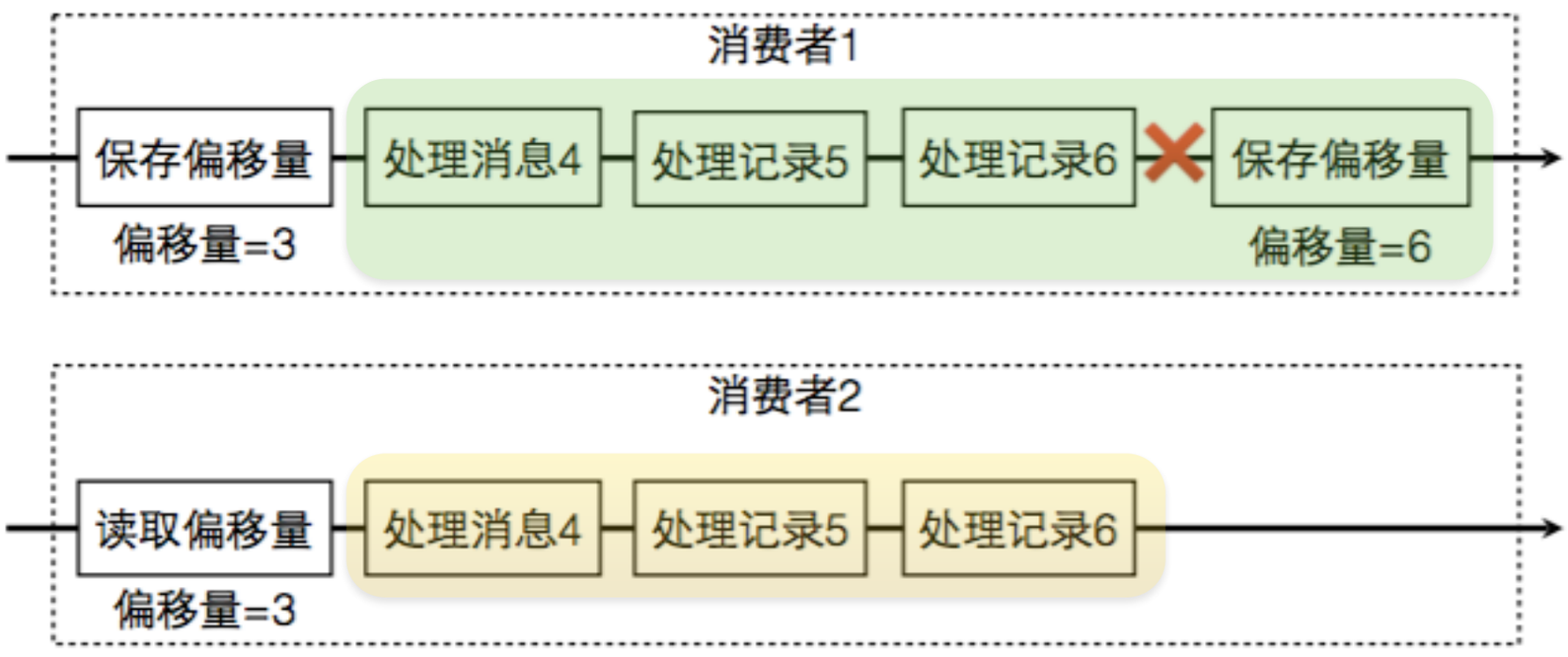
消息传递语义

至多一次：先提交后消费



消费者1挂掉，消费者2从偏移量=3的位置开始消费，丢失了记录2和记录3

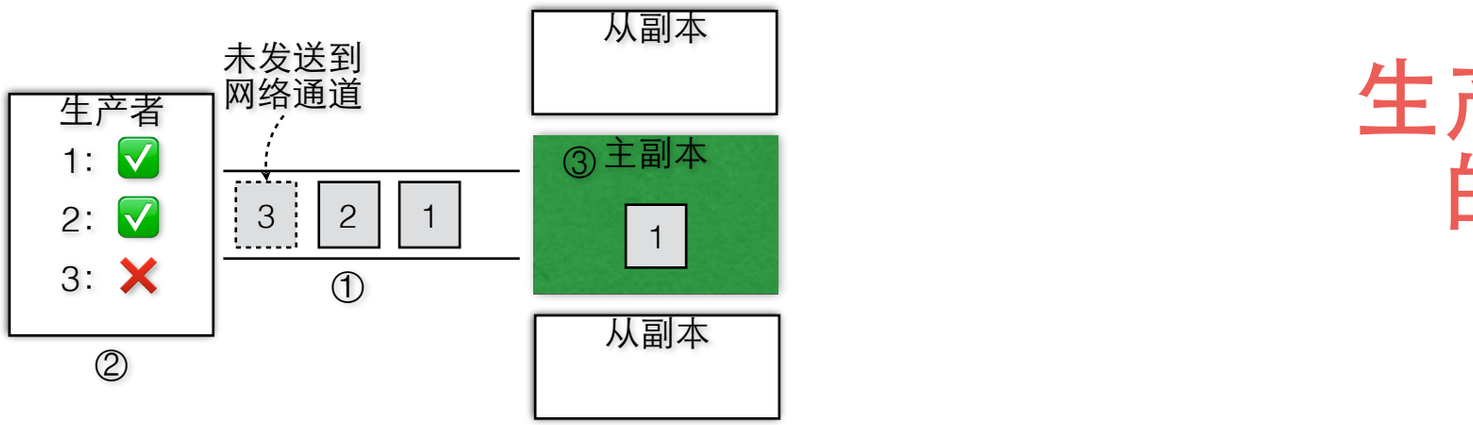
至少一次：先消费后提交



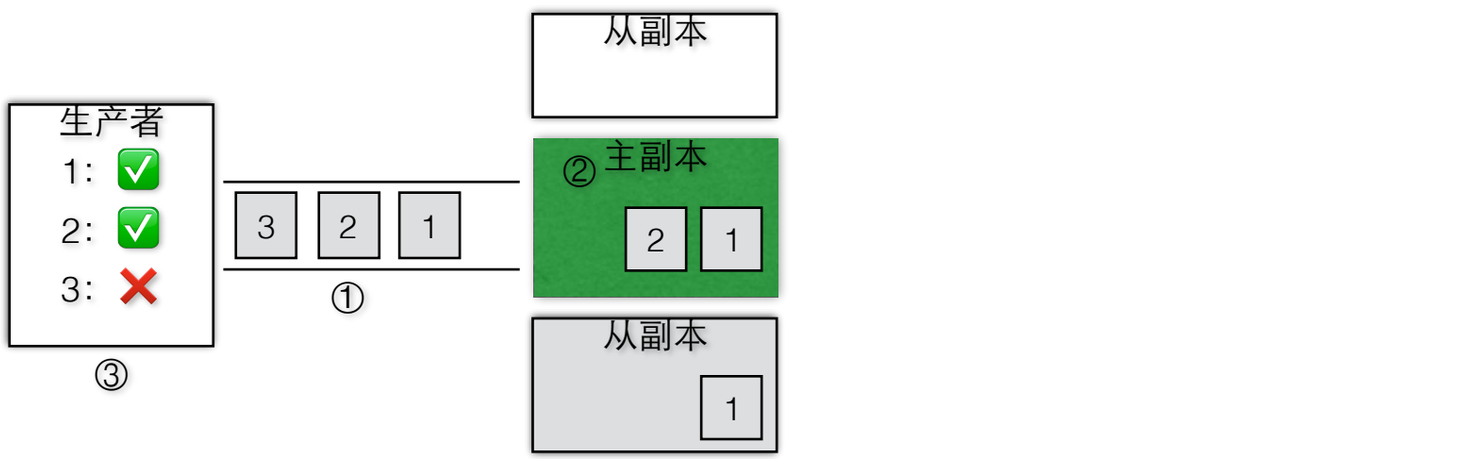
消费者1挂掉，消费者2从偏移量=3的位置开始消费，重复了记录4、5、6

生产者数据丢失的一种示例

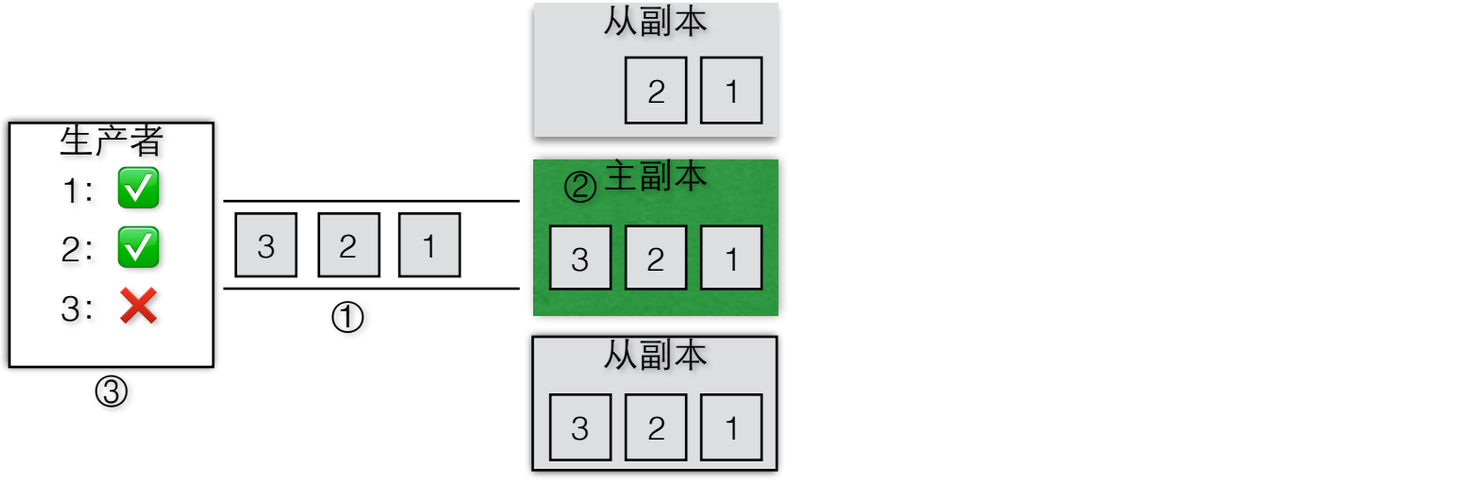
acks=-1



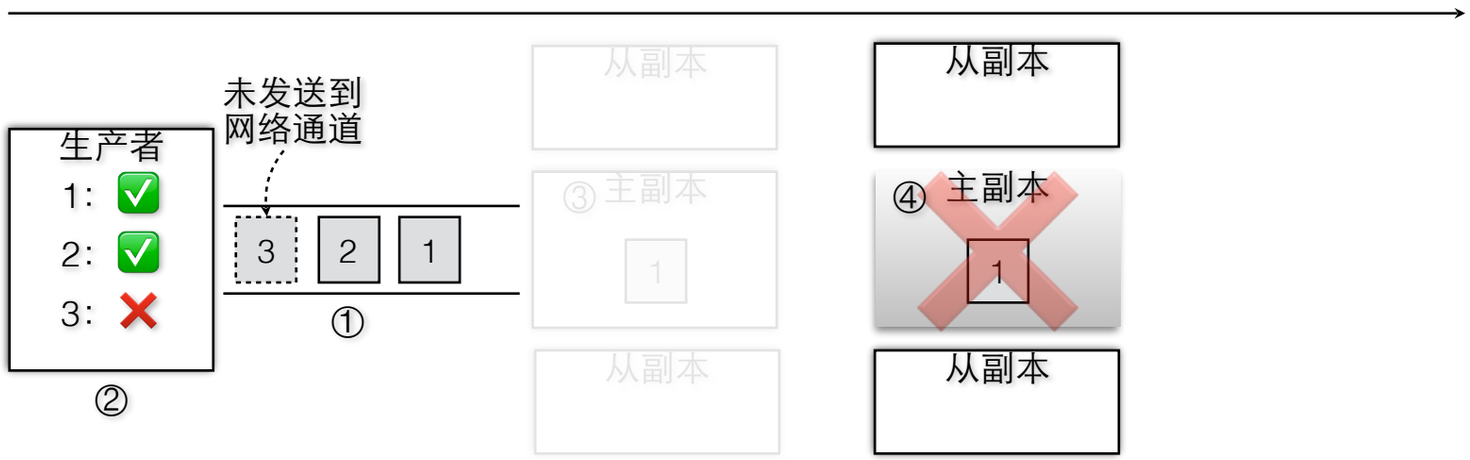
acks=1



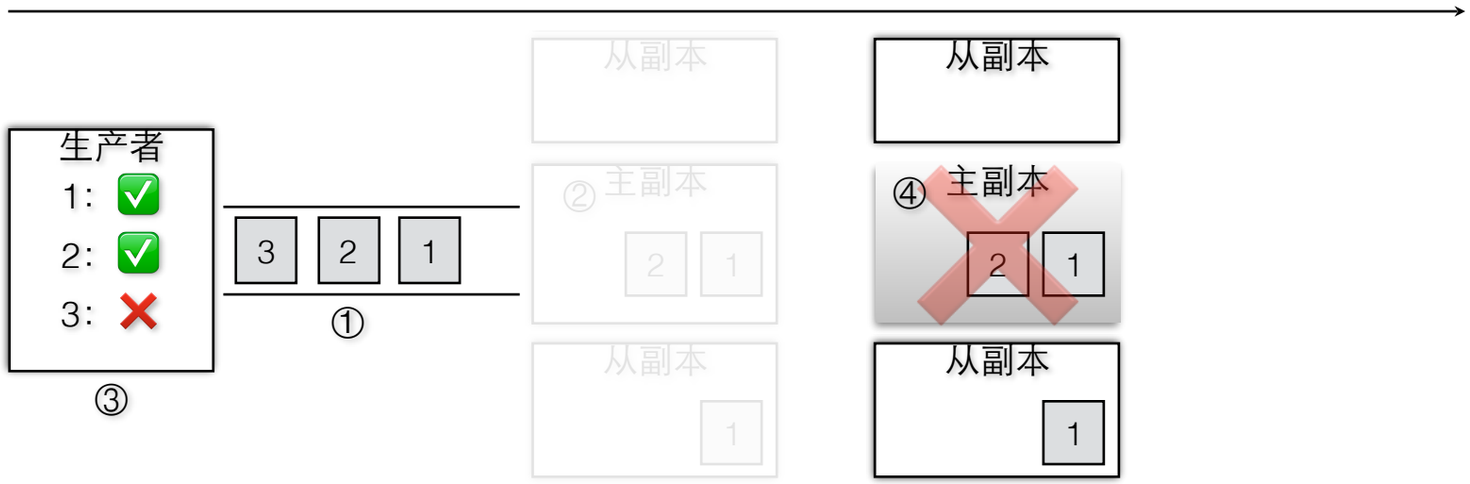
acks=all



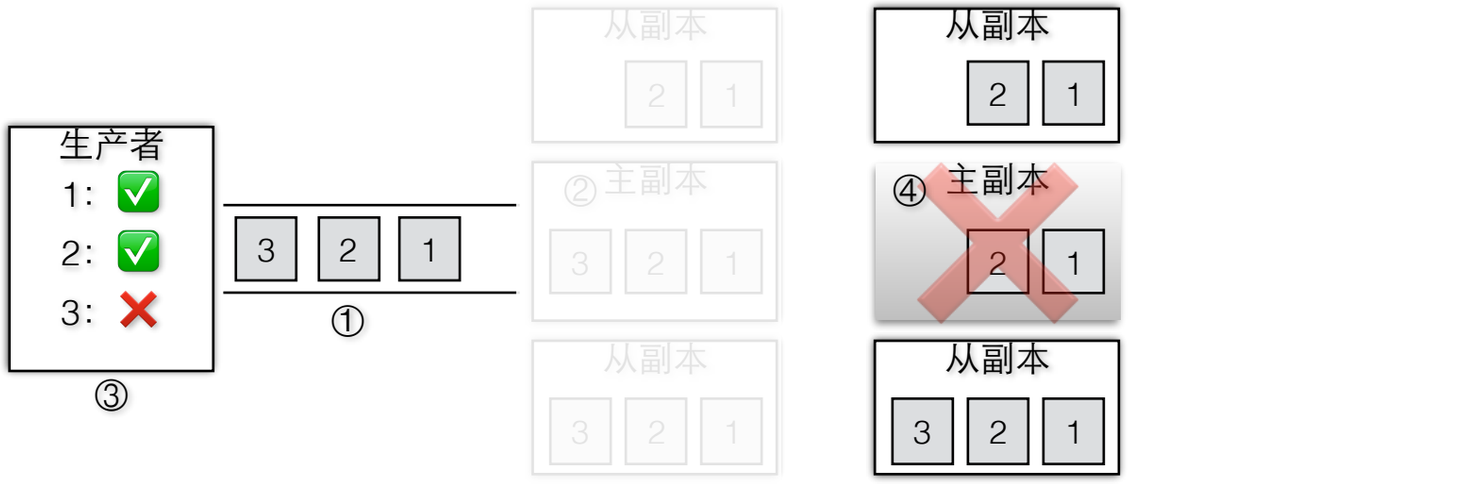
acks=-1

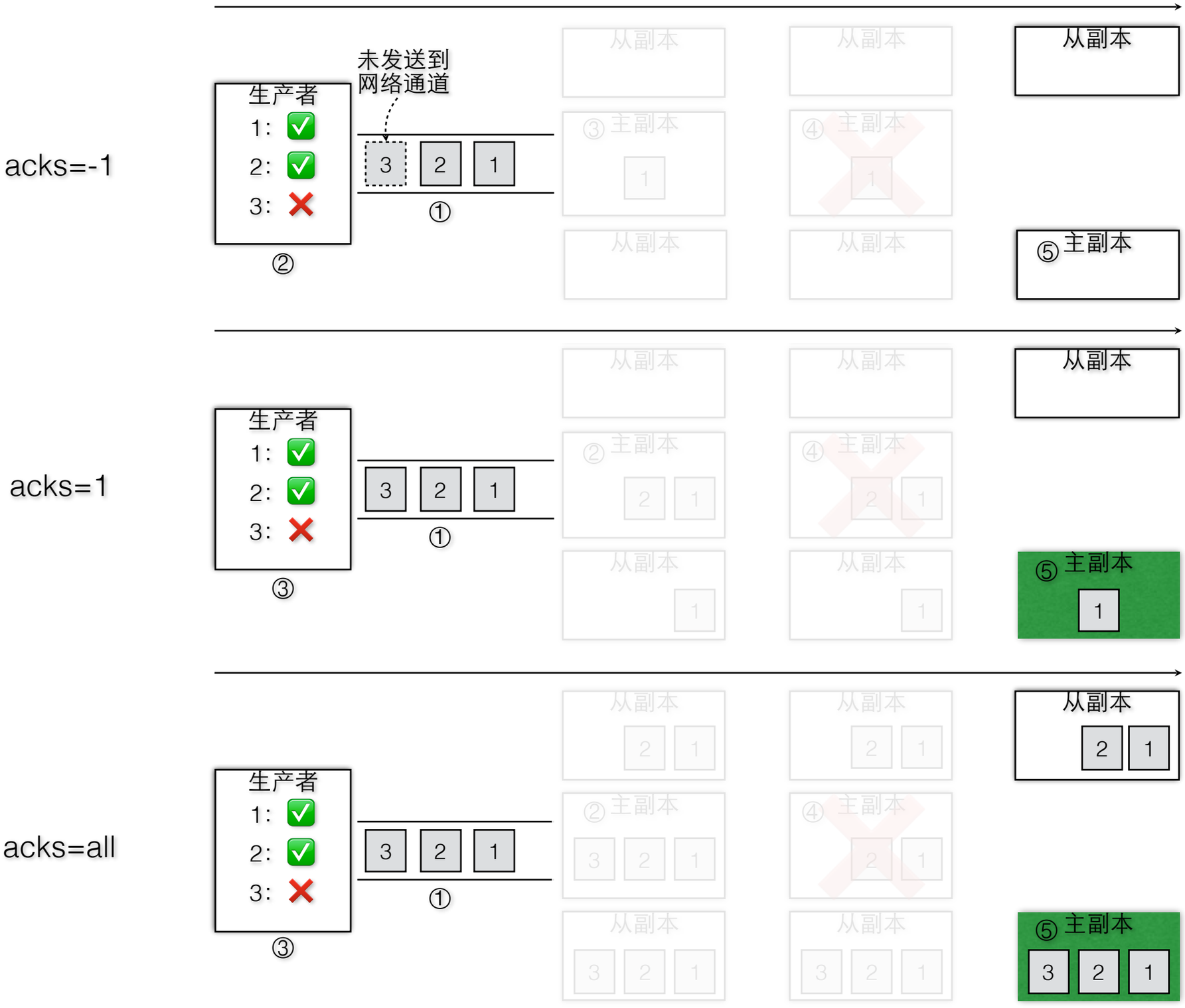


acks=1



acks=all



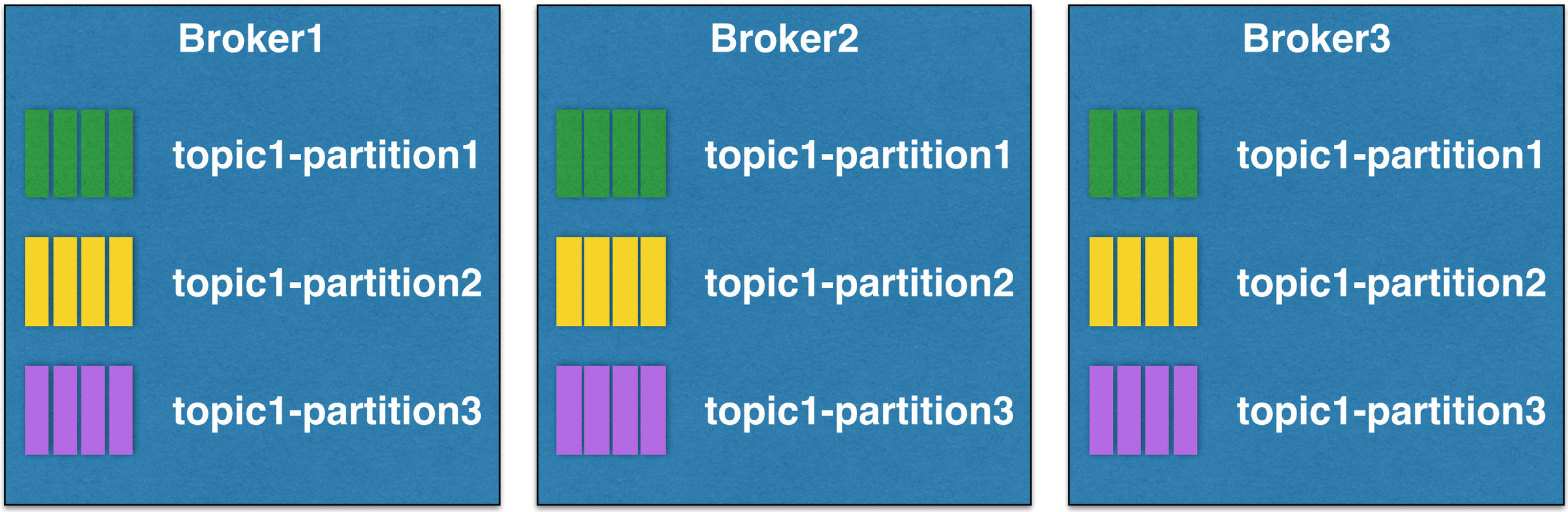


设置broker的min.insync.replicas=2，否则生产者即使acks=all，也有可能丢失数据。因为isr是动态调整的。

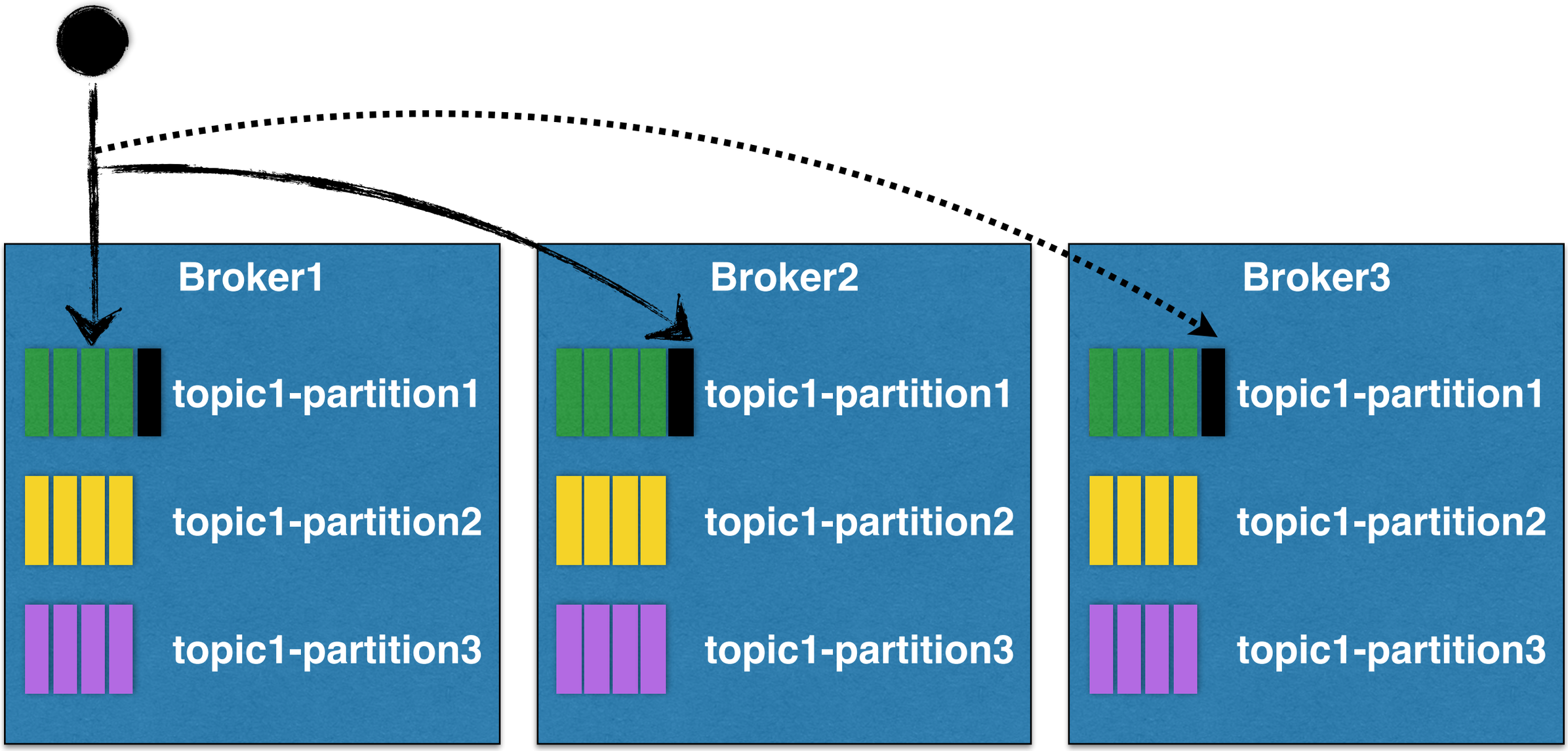
Broker故障

为什么需要副本机制?

Replications



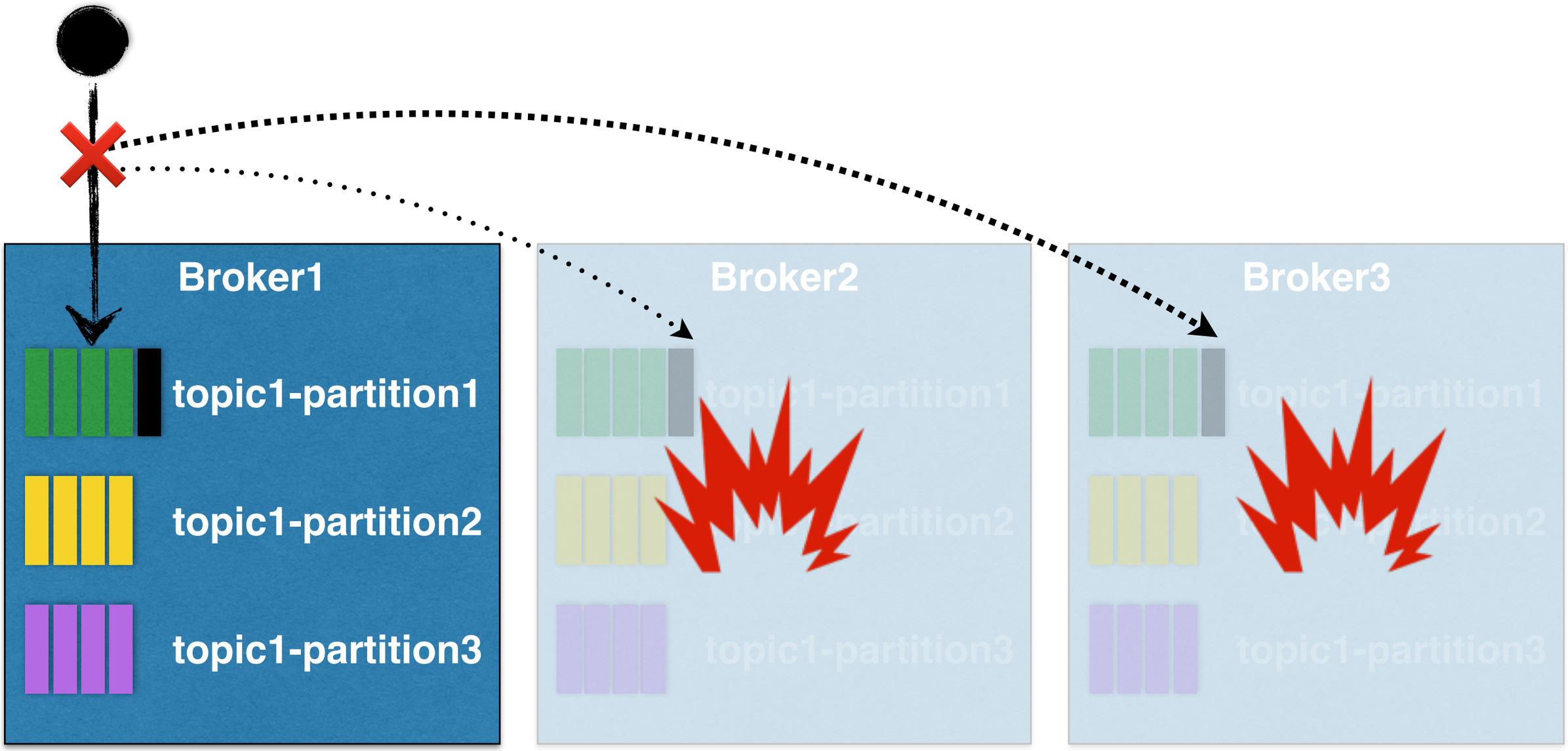
Replication with Quorum



3个副本，最多允许挂一台。如果挂两台，失败

至少Replicator(3)/2+1=2台存活

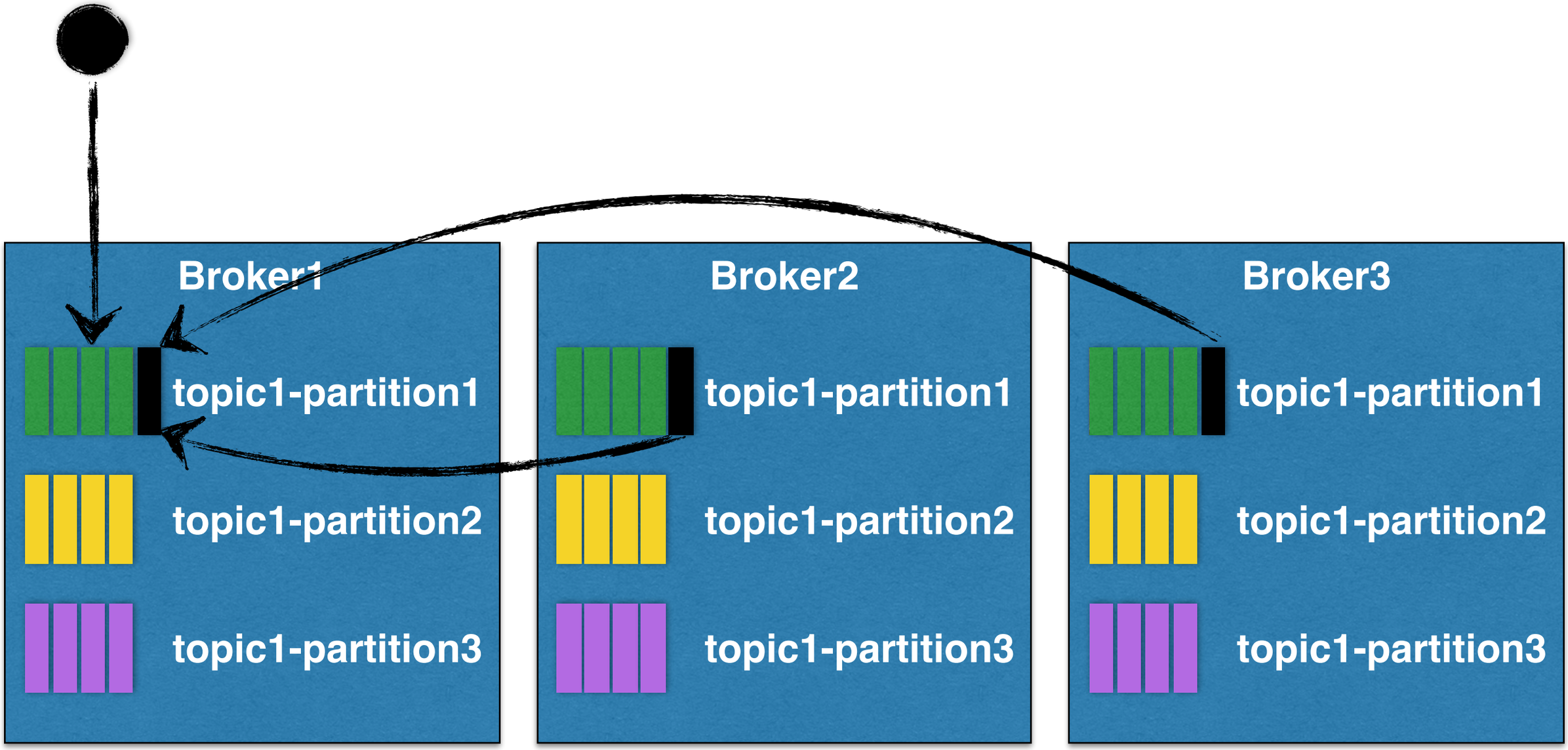
Replication with Quorum



3个副本，最多允许挂一台。如果挂两台，失败

至少Replicator(3)/2+1=2台存活

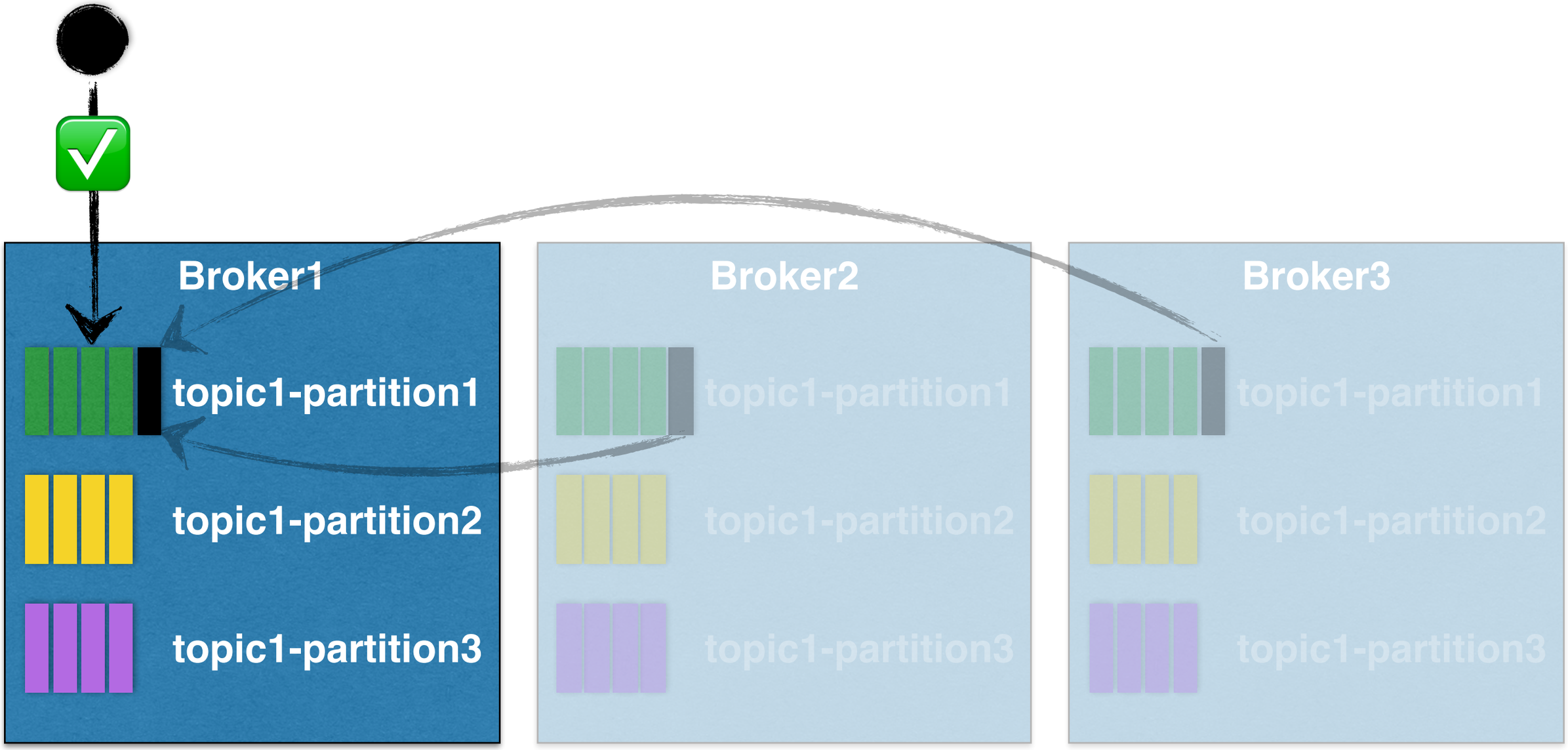
Replication with ISR



3个副本，最多允许挂两台。容错性更高

只要一台存活就OK

Replication with ISR

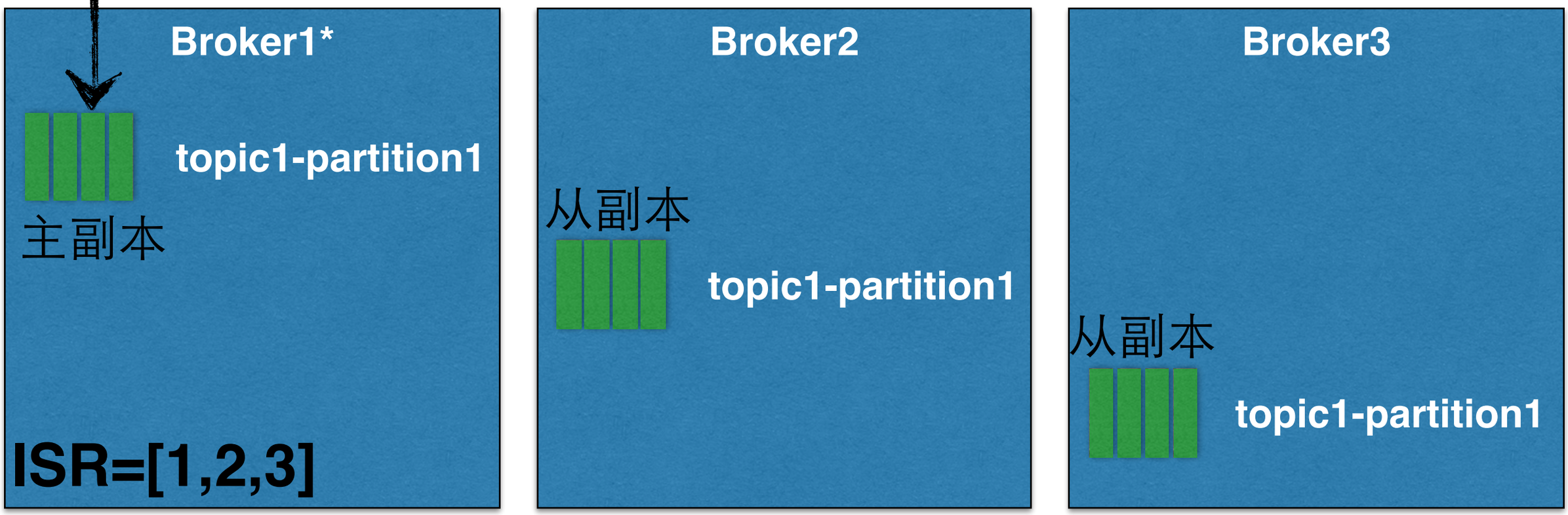


3个副本，最多允许挂两台。容错性更高

只要一台存活就OK

ack with all ISRS

write(ack=all)

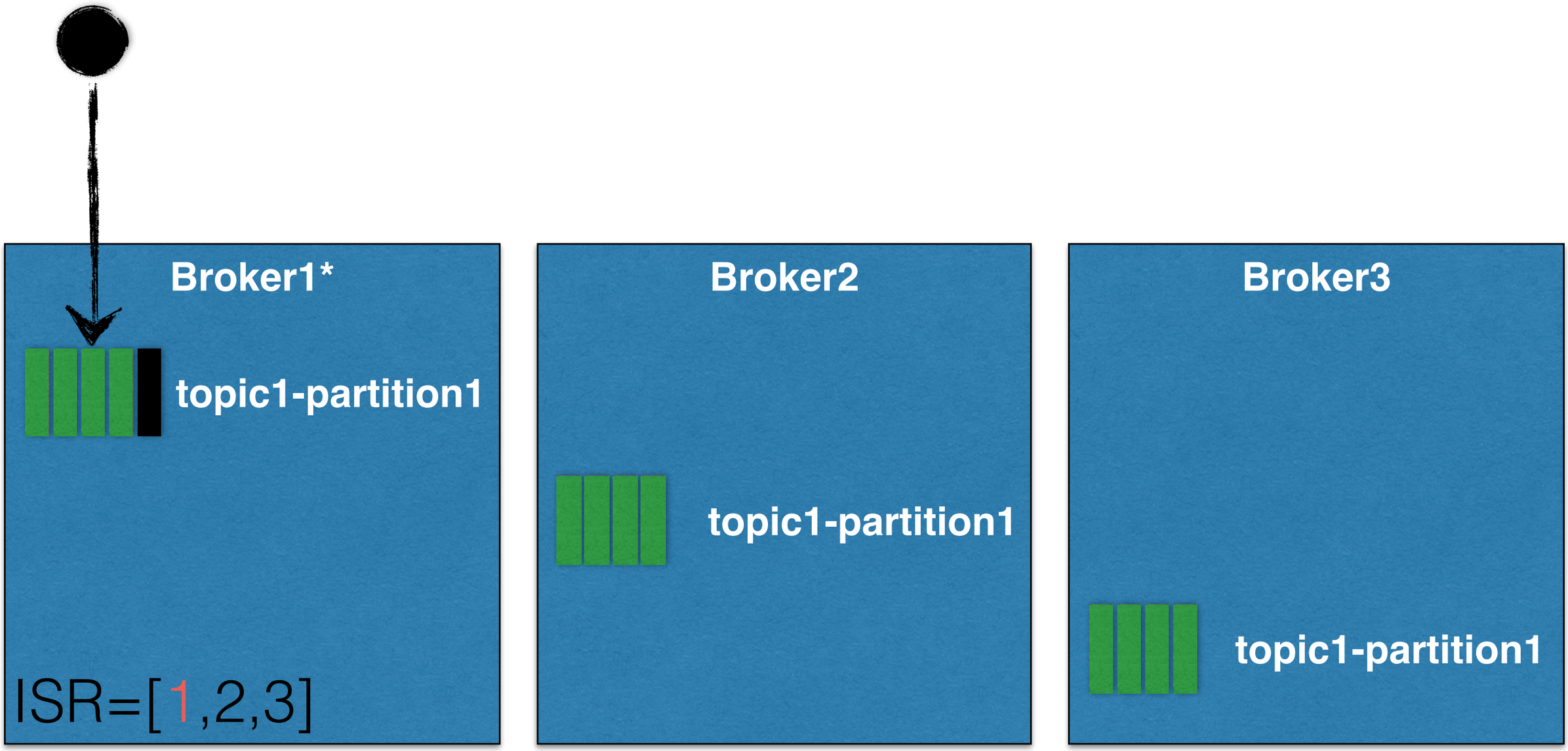


主副本记录了分区的ISR（in-sync-replicas）信息

对于落后太多的副本，主副本可以将其从ISR中移除

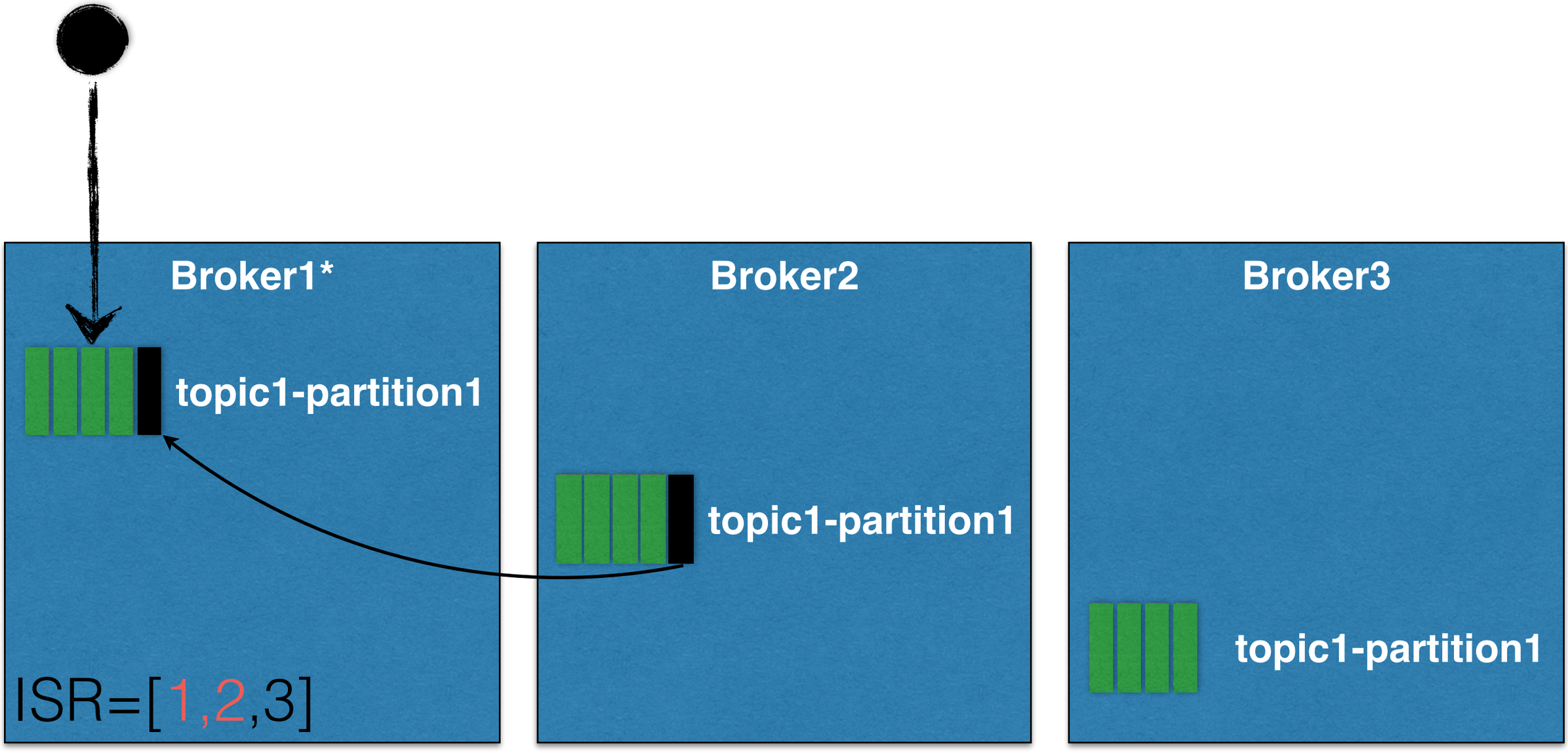
ack with all ISRS

write(ack=all)



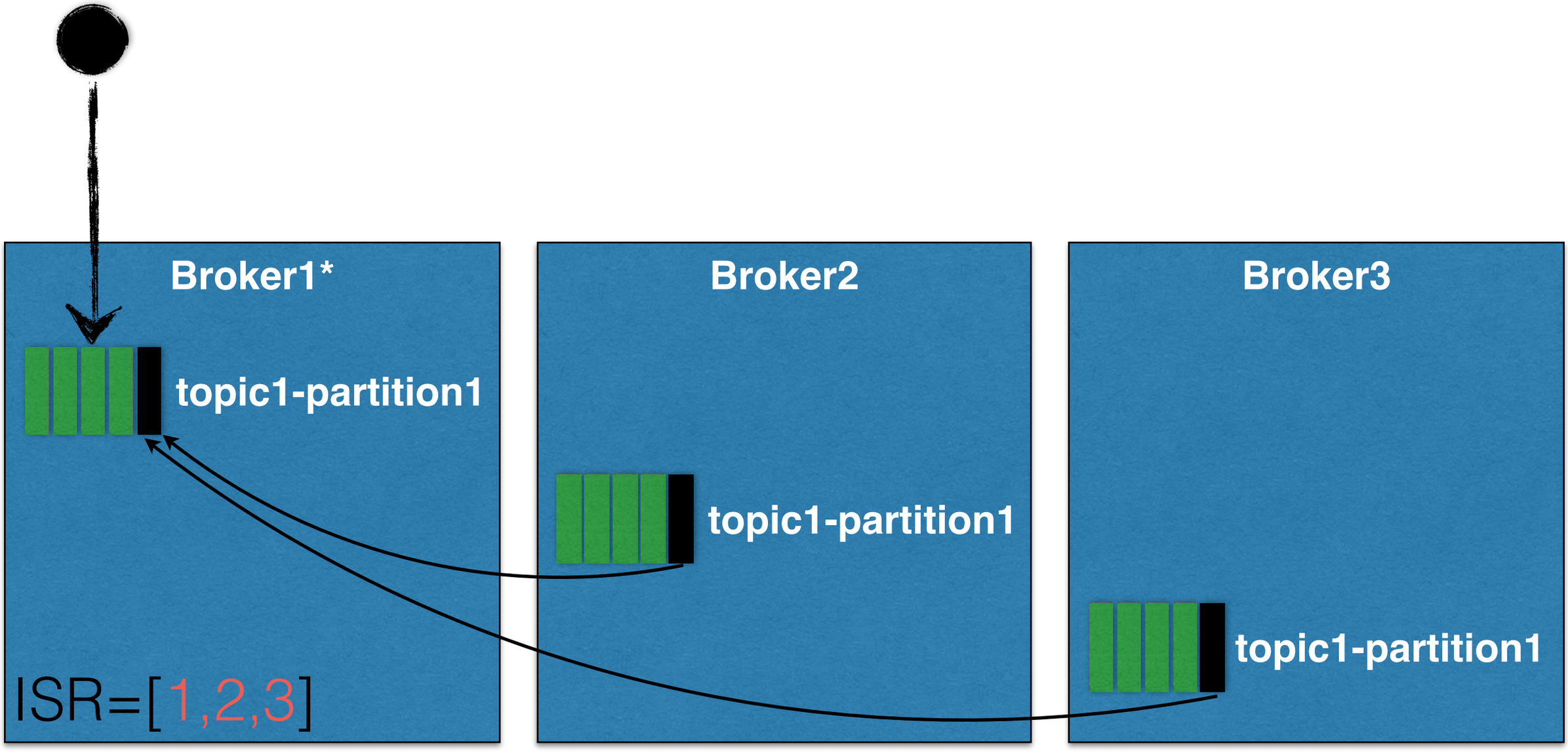
ack with all ISRS

write(ack=all)



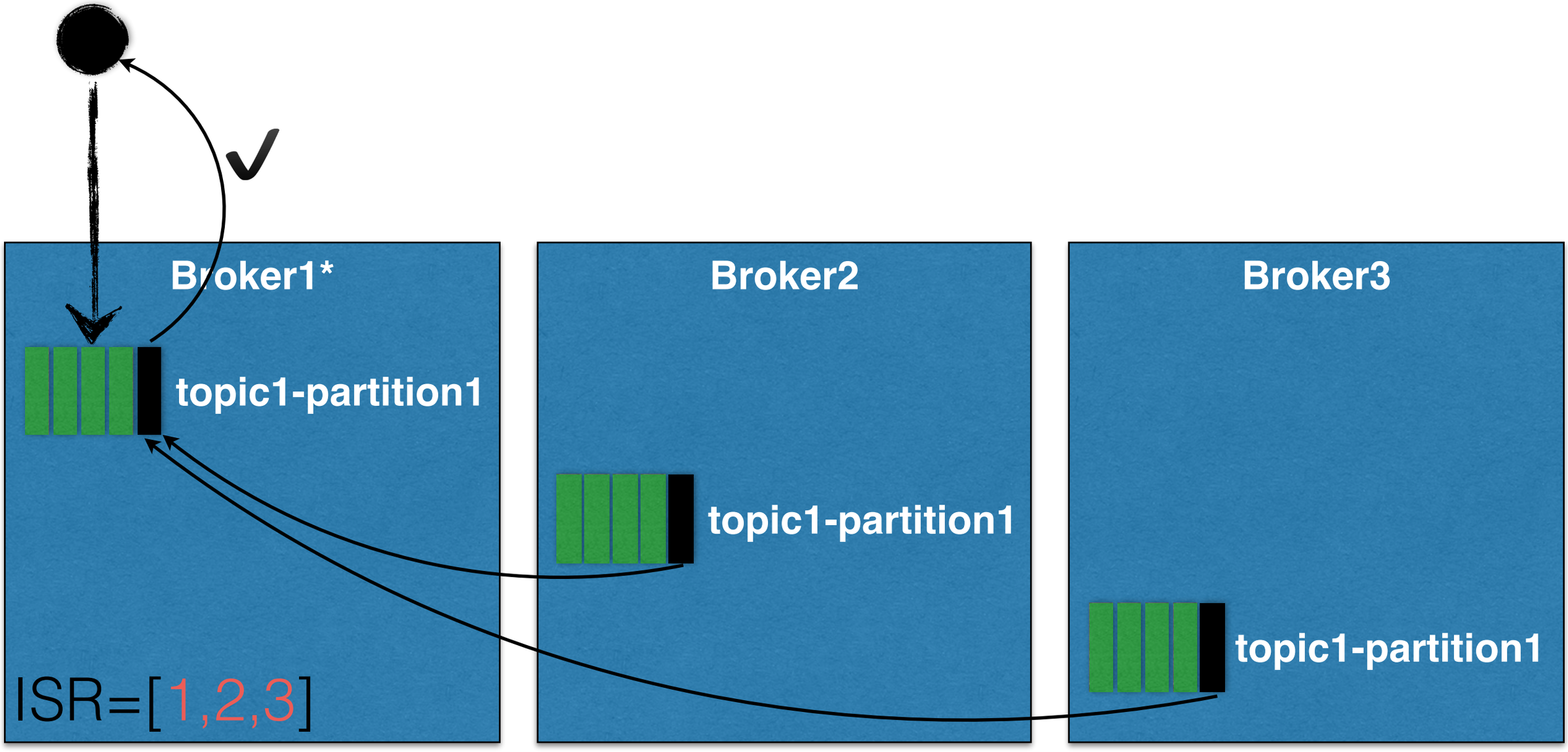
ack with all ISRS

write(ack=all)



ack with all ISRS

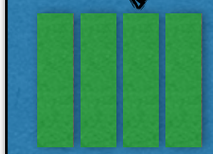
write(ack=all)



ack with leader only

write(ack=1)

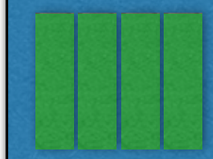
Broker1*



topic1-partition1

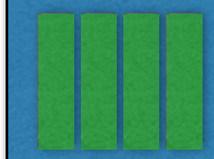
ISR=[1,2,3]

Broker2



topic1-partition1

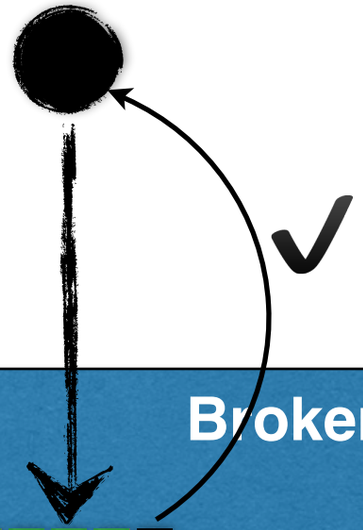
Broker3

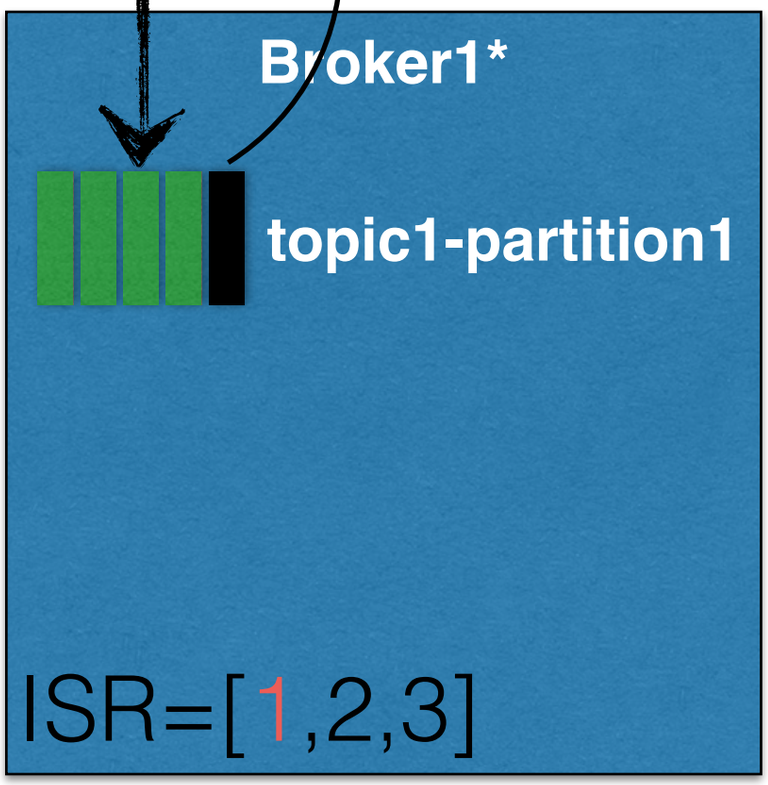


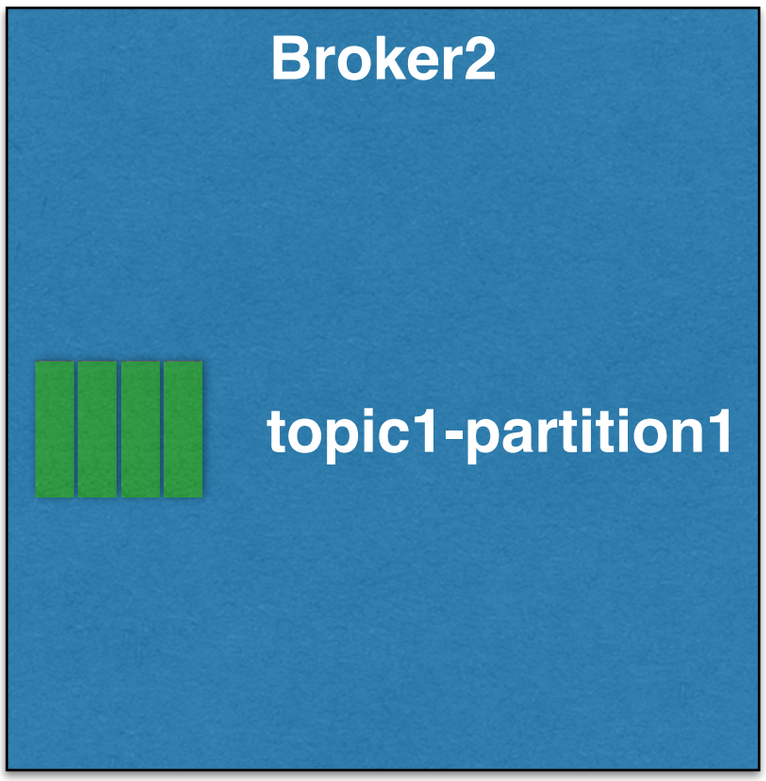
topic1-partition1

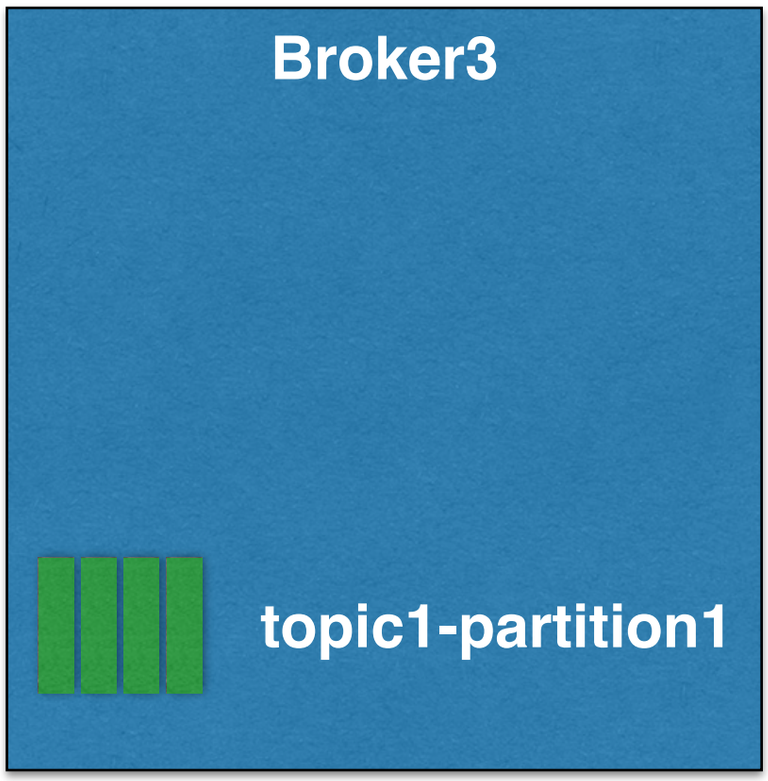
ack with leader only

write(ack=1)

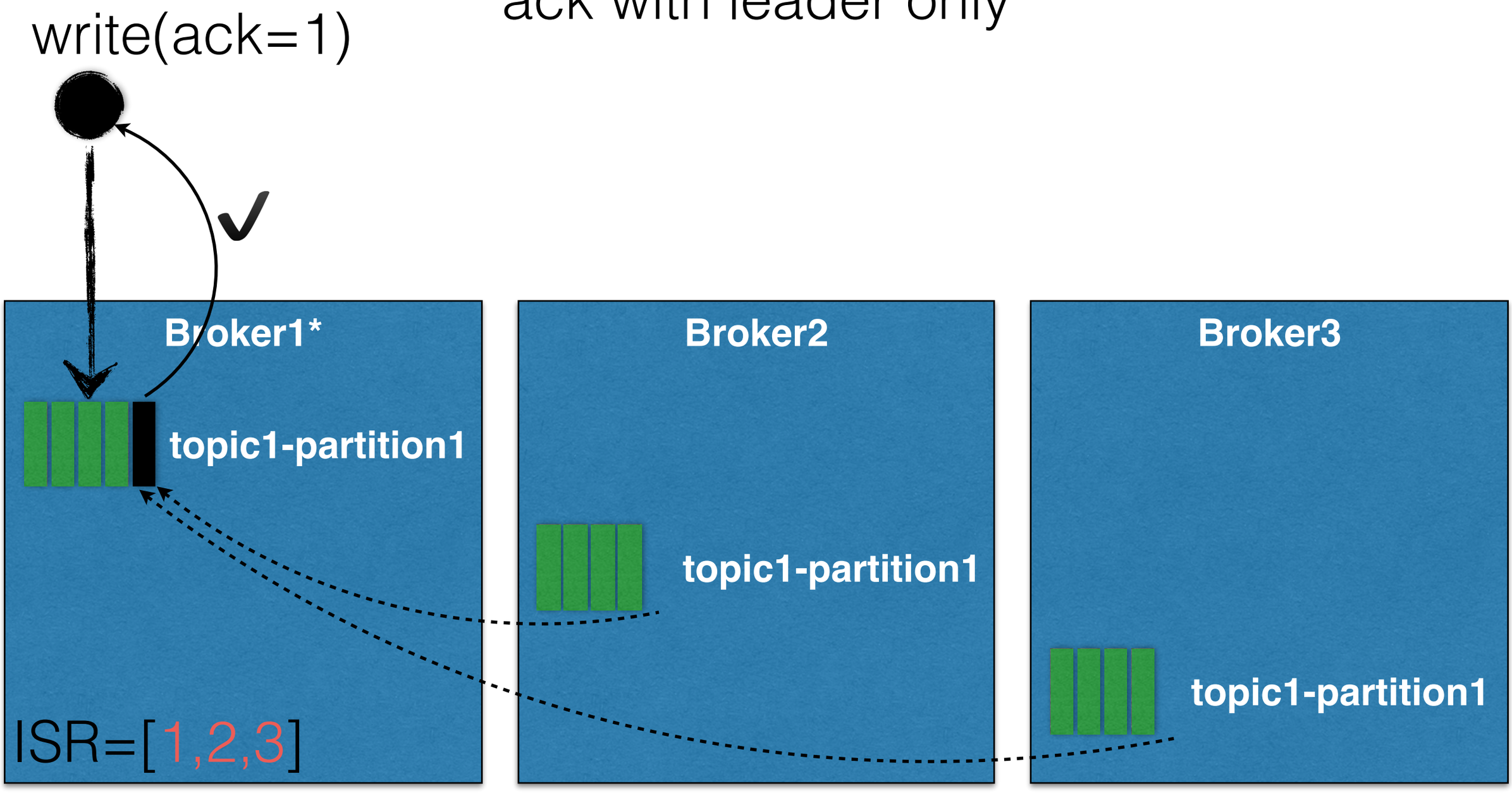






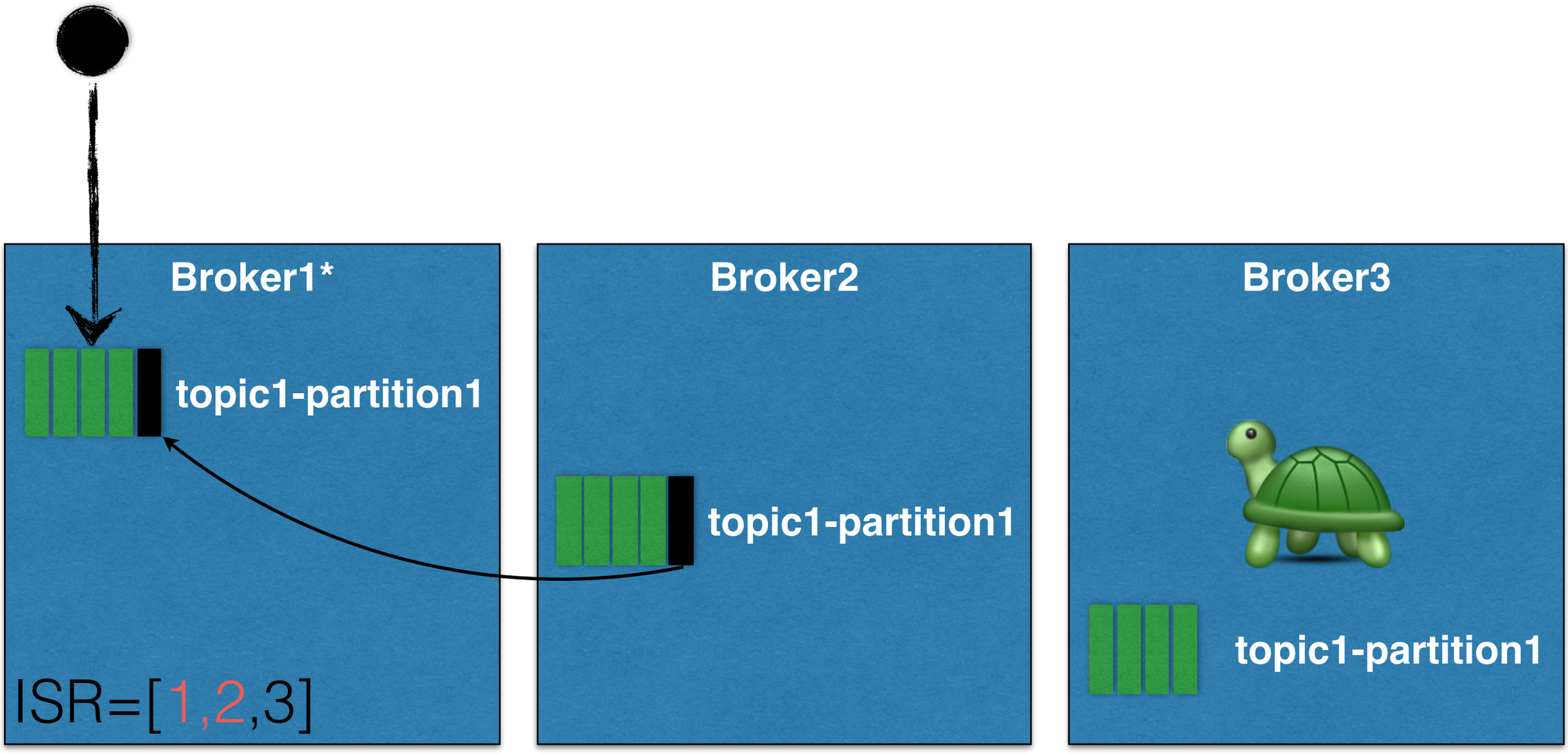


ack with leader only

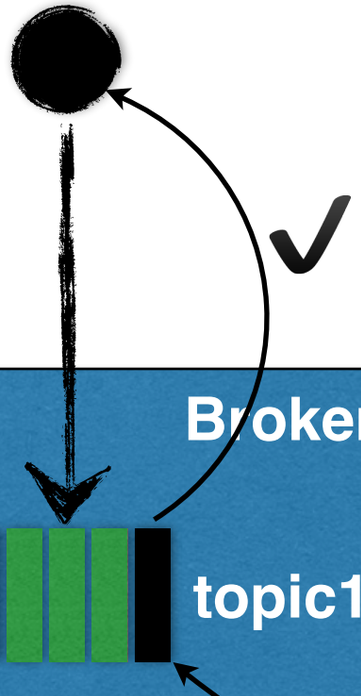


slow follower

write(ack=all)



write(ack=all)



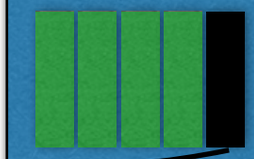
slow follower

Broker1*

topic1-partition1

ISR=[1,2]

Broker2

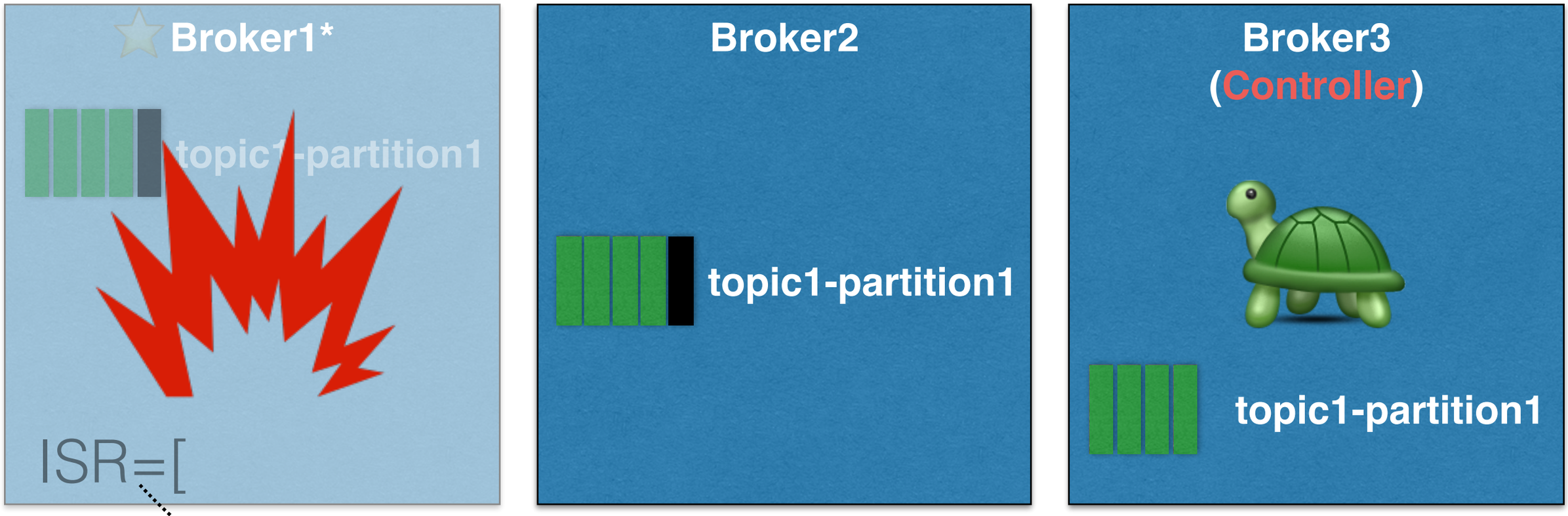


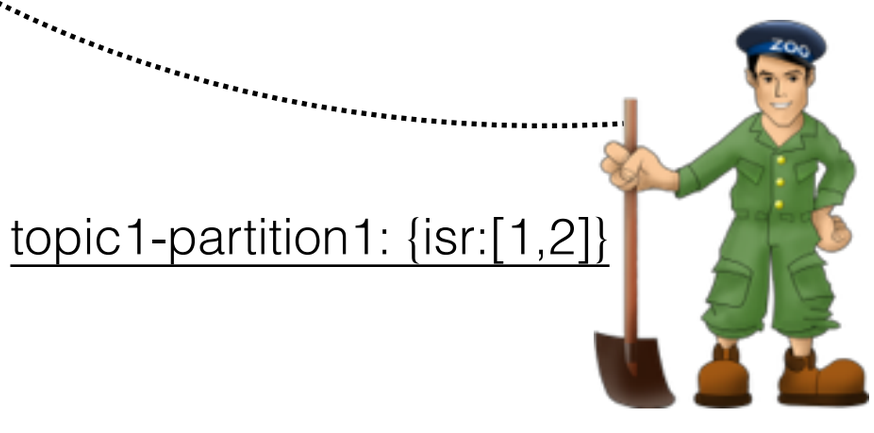
topic1-partition1

Broker3

topic1-partition1

Leader down


Broker1*
topic1-partition1
ISR=[
Broker2
topic1-partition1
Broker3
(Controller)
topic1-partition1


topic1-partition1: {isr:[1,2]}

New Leader

Broker1*

topic1-partition1

ISR=[

Broker2*

主副本

topic1-partition1

ISR=[**2**]

Broker3
(Controller)

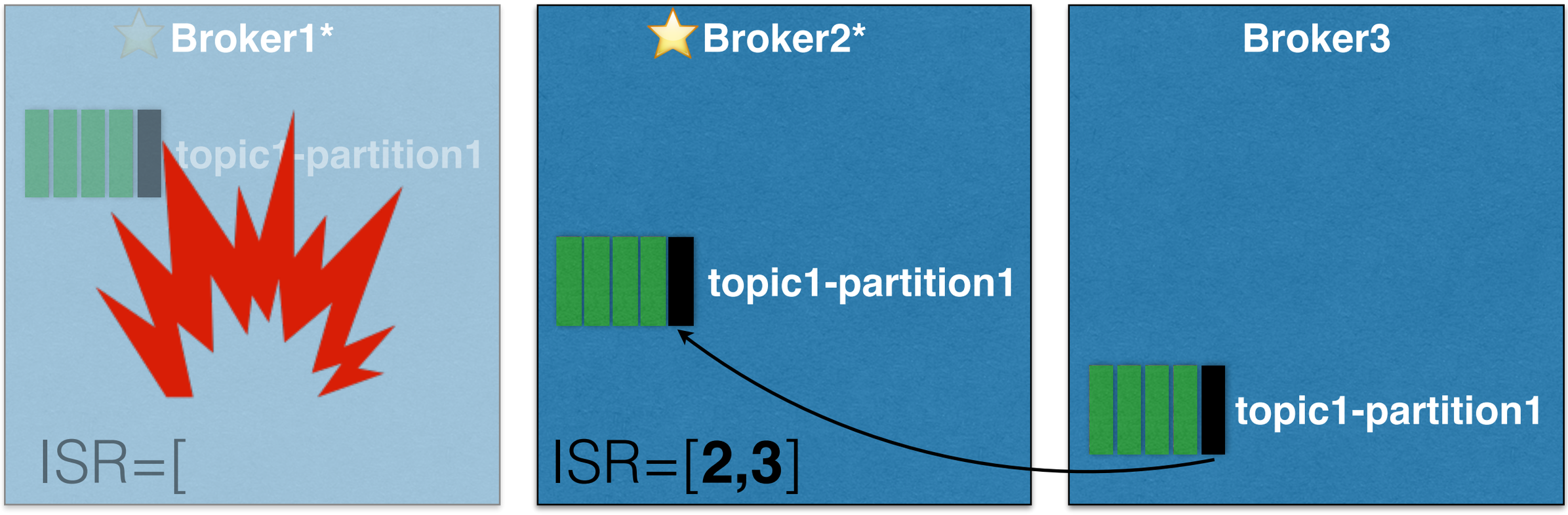
从副本

topic1-partition1

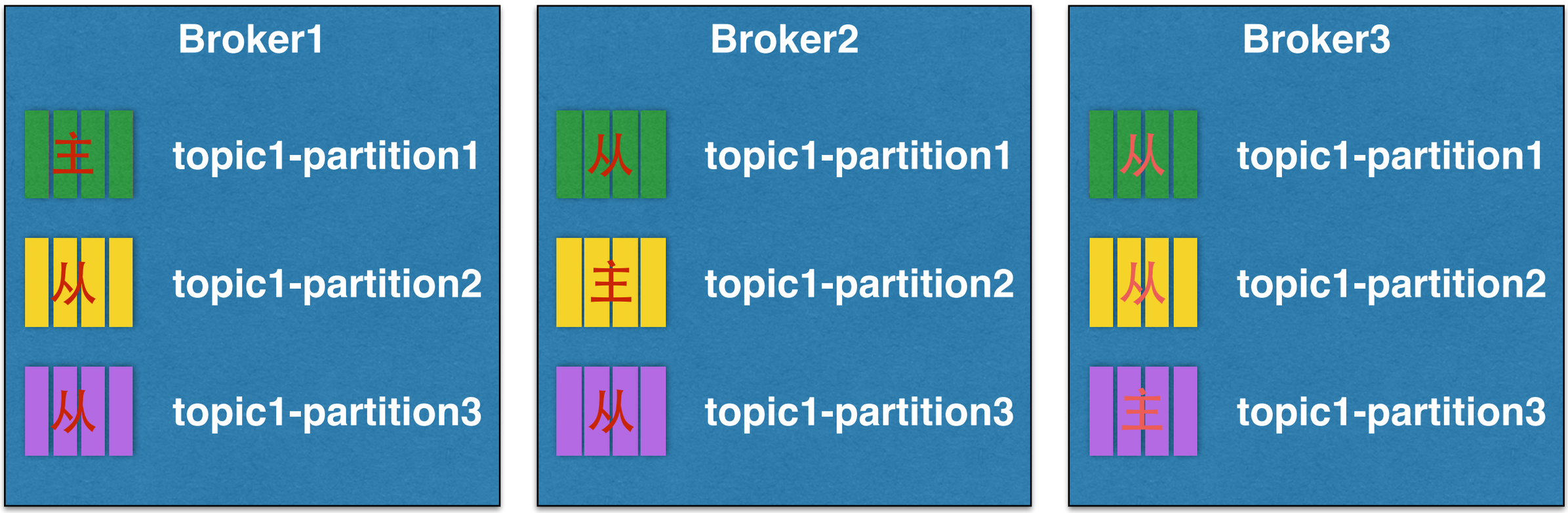
ISR=[**2**]

topic1-partition1: {isr:[1,**2**]}

new leader


Broker1*
topic1-partition1
ISR=[
Broker2*
topic1-partition1
ISR=[2,3]
Broker3
topic1-partition1

每个Broker是某些分区的主副本，
同时又是其他一些分区的从副本。
保证分区的主副本均匀地落在集群的每个Broker上。



客户端（生产者和消费者）只读写分区的主副本。
从副本仅用做故障发生时，重新选举新的主副本。

replication example

ISR示例:

broker1

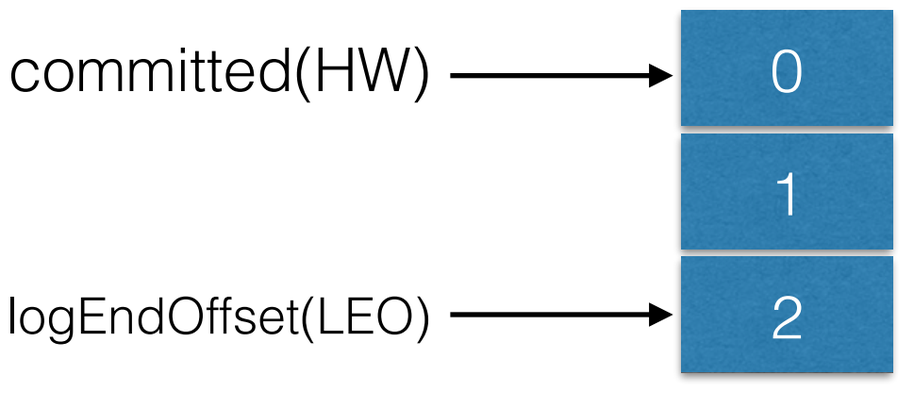
leader
isr=[1,2,3]

broker2

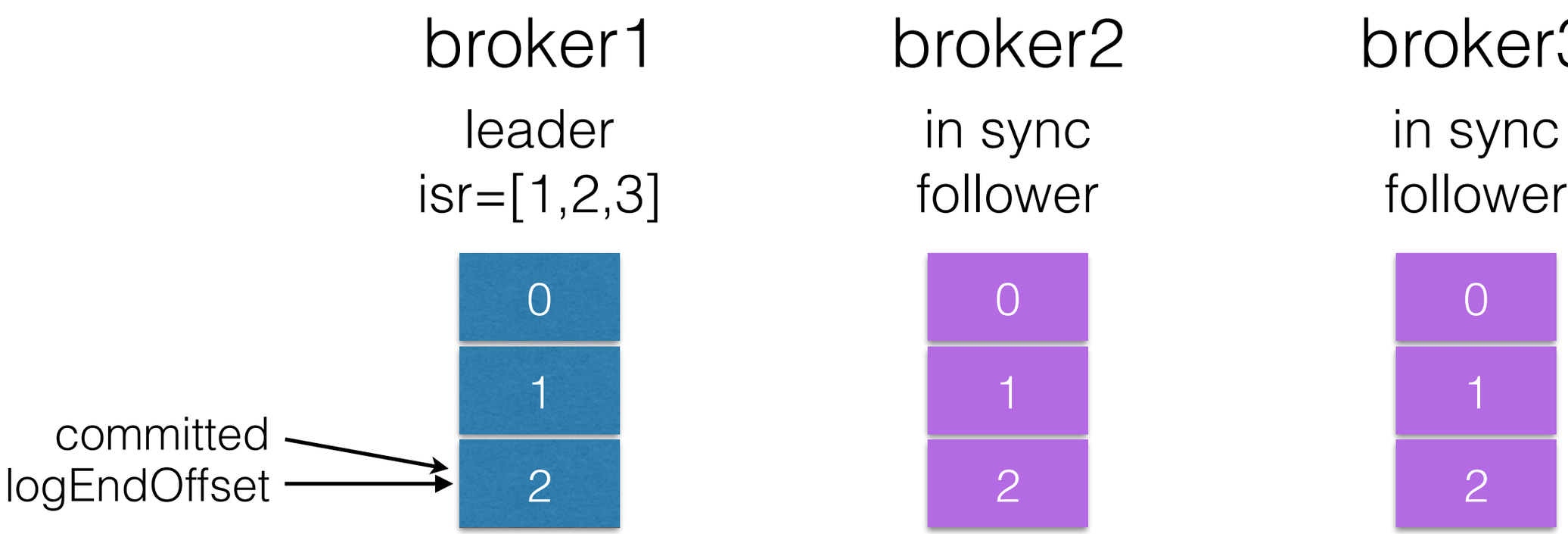
follower1

broker3

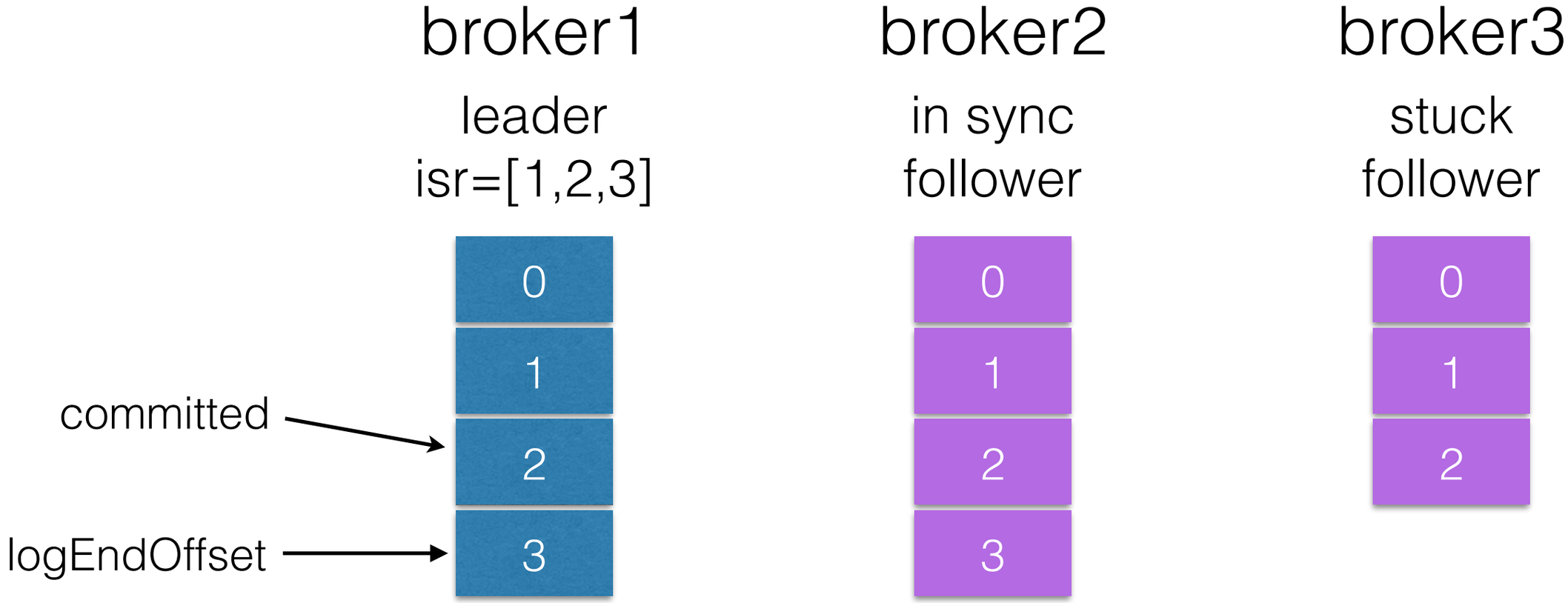
follower2



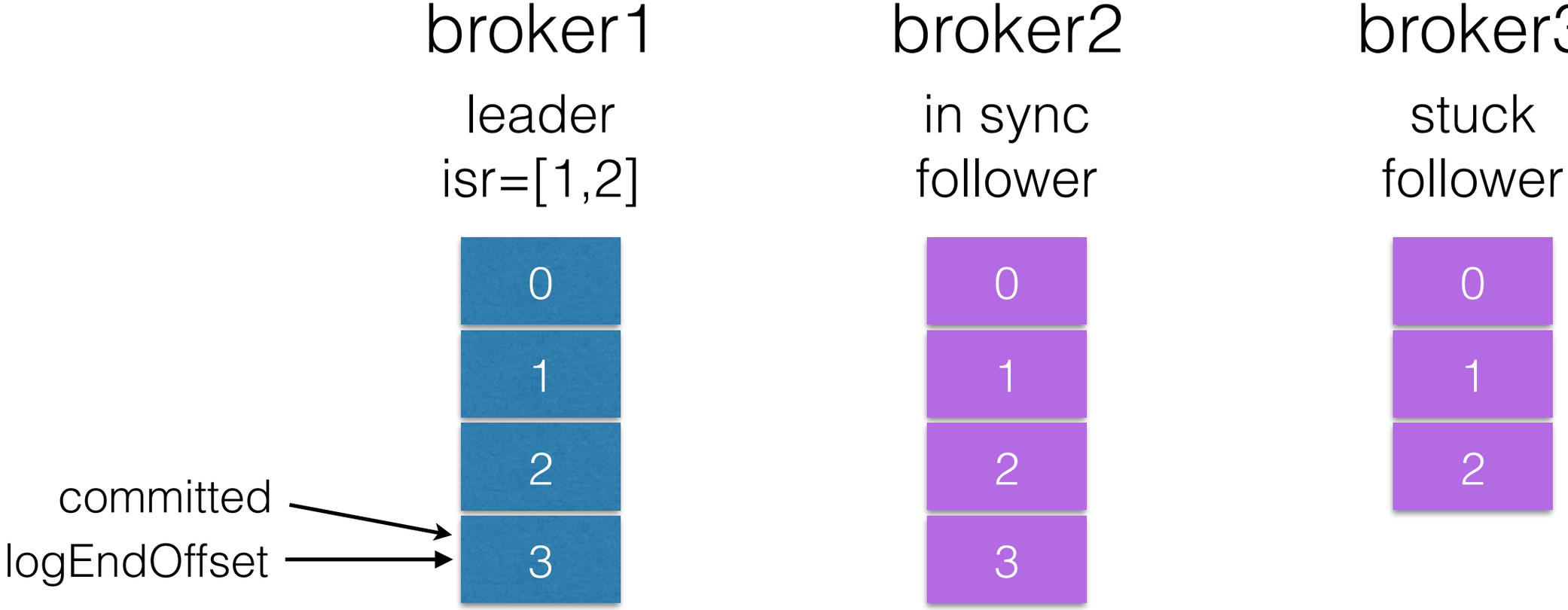
replication example



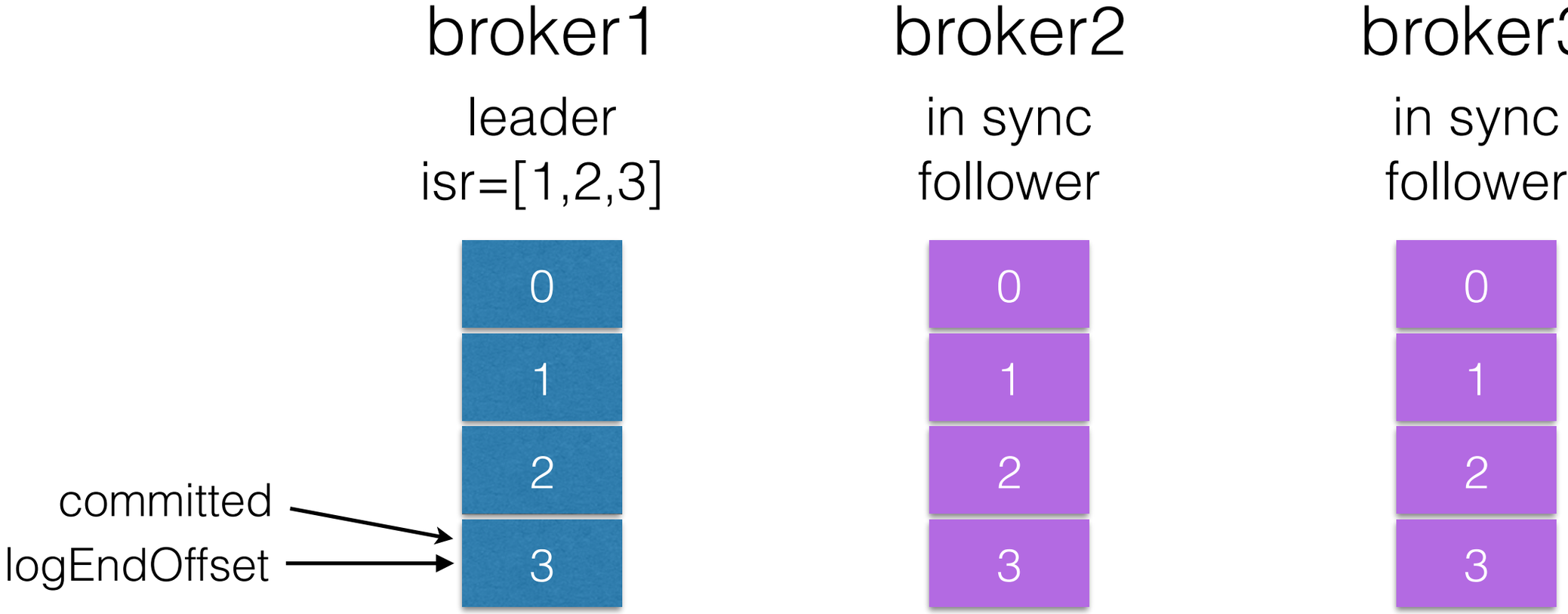
replication example



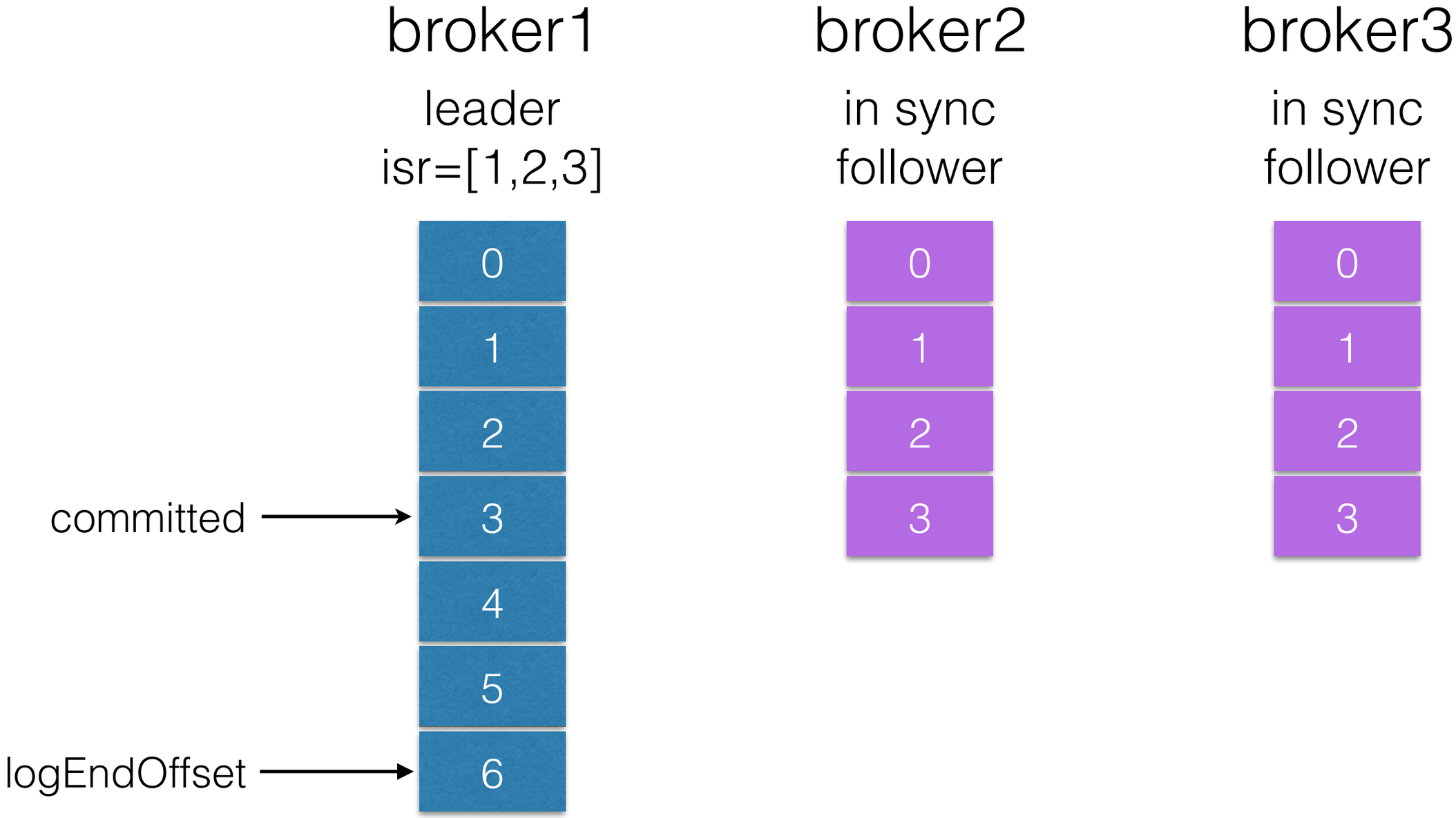
replication example



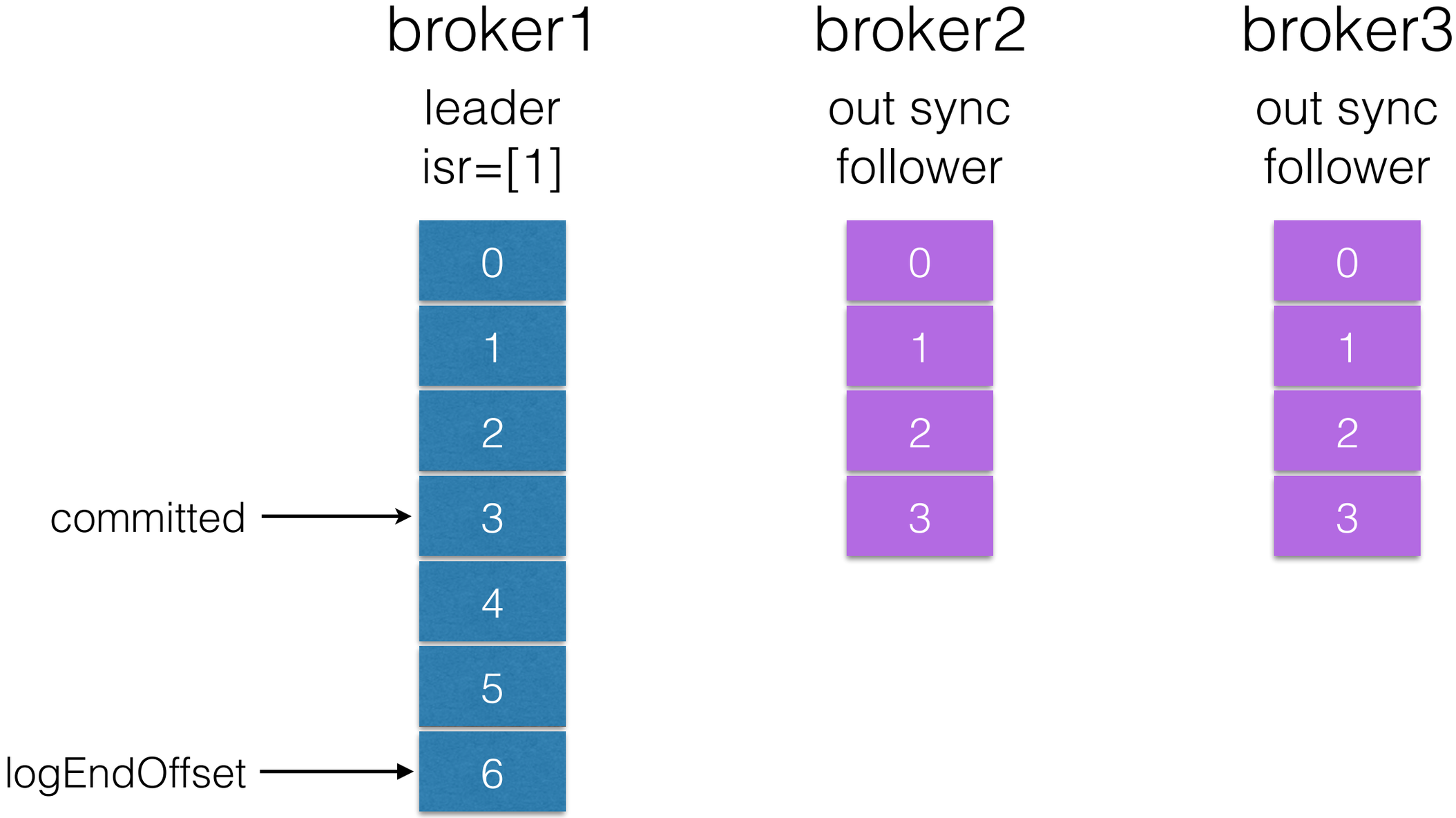
replication example



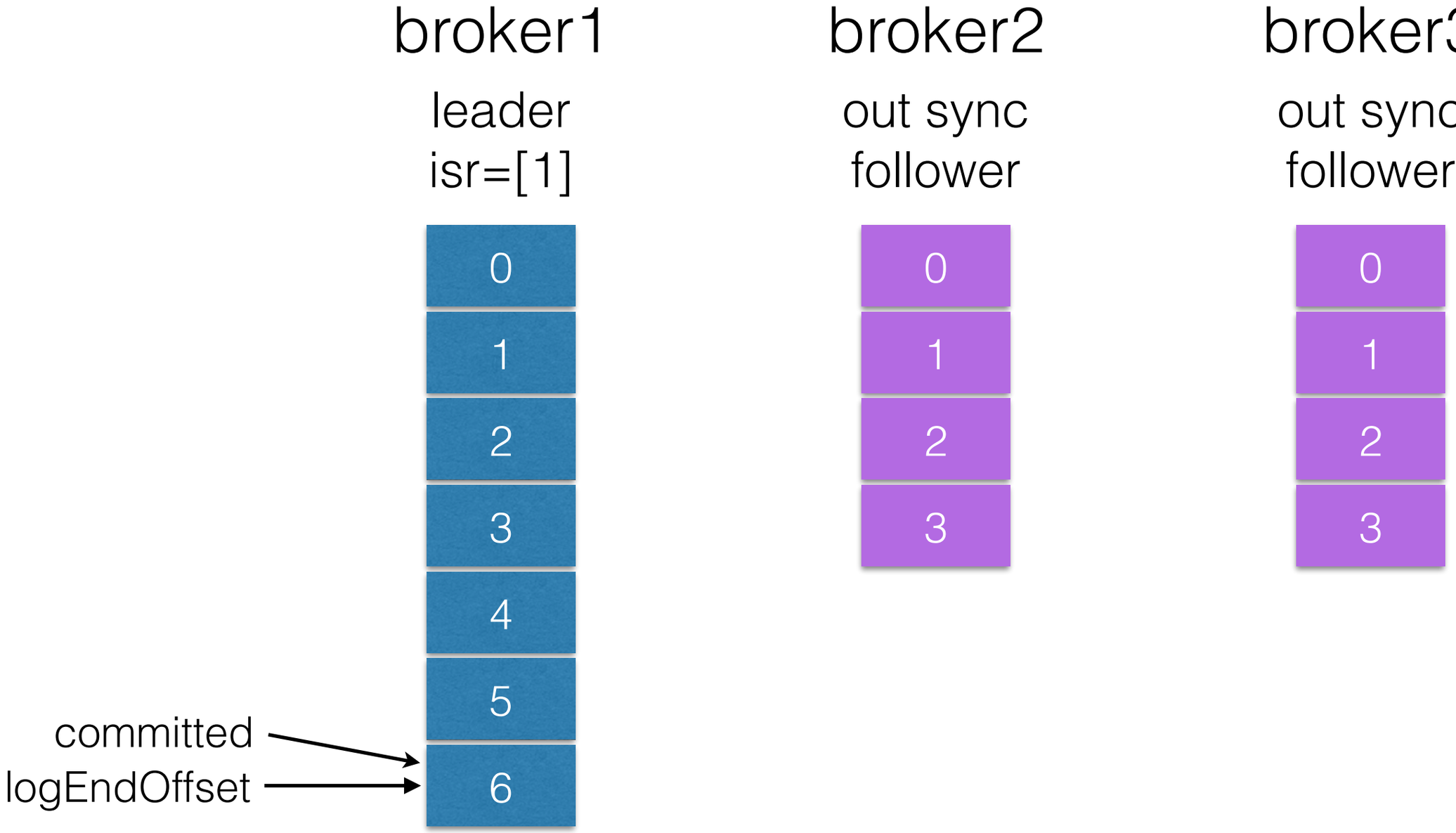
replication example



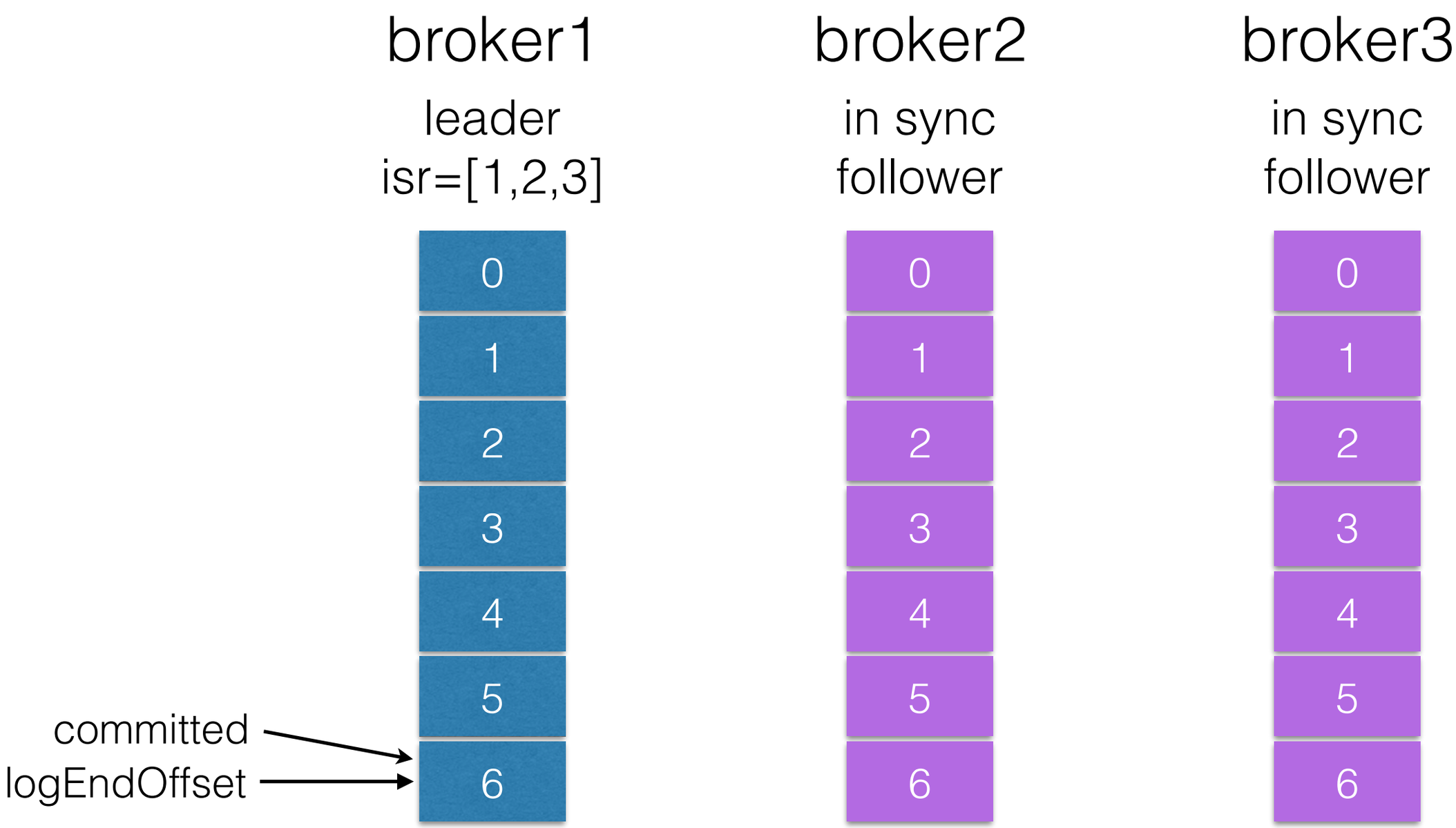
replication example



replication example



replication example



如果没有及时发送拉取请求（replica.lag.time.max.ms=10s），会被剔除

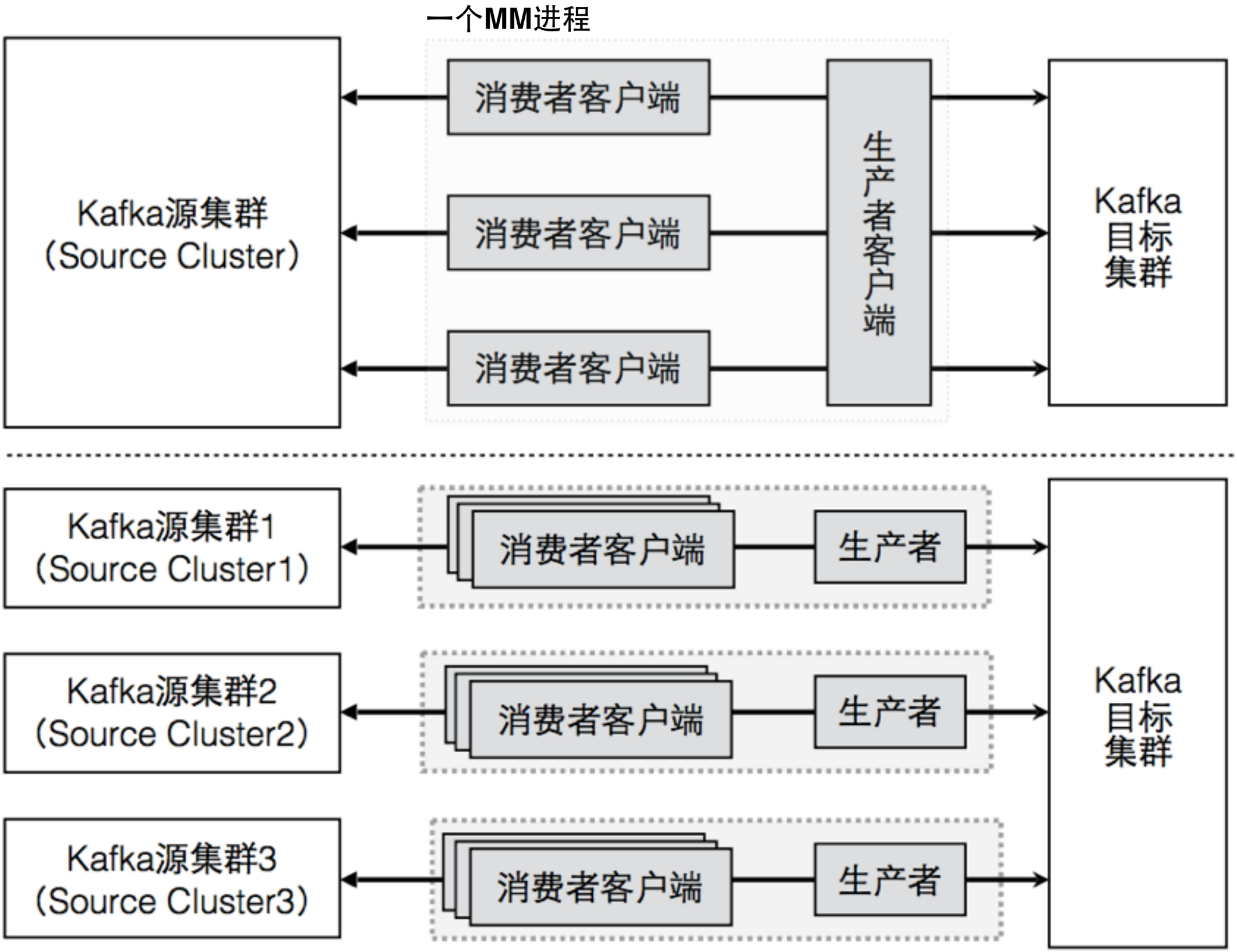
还有一个条件：即使有拉取请求，但是没有赶上Leader，也会被剔除

消费者的一个核心概念：消费组管理协议

一个分区只被同一个消费组中的一个消费者使用：保证顺序

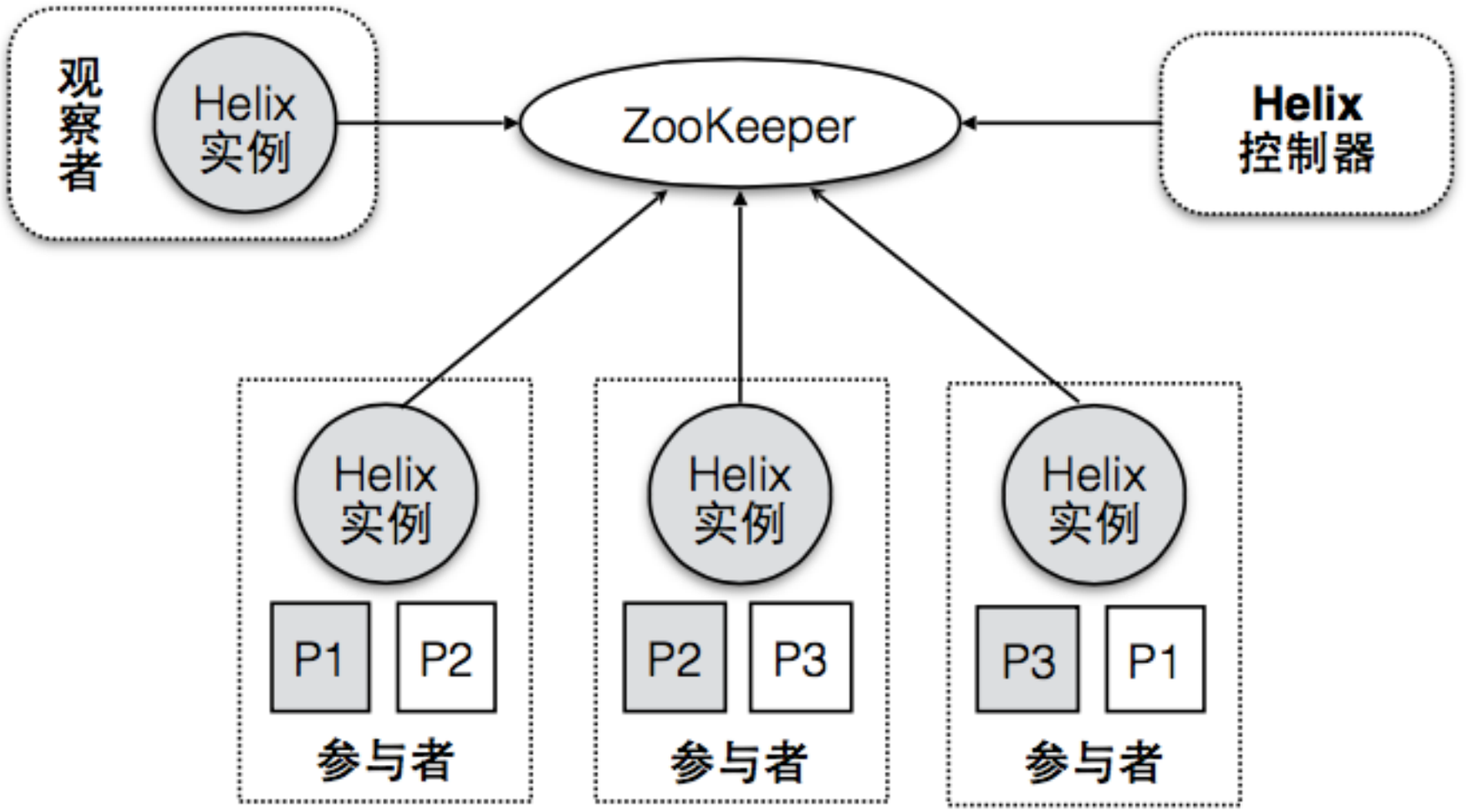
一个主题可以被同一个消费组的多个消费者订阅：扩展性

MirrorMaker



MM利用了Kafka的消费组管理协议

问题：为什么要采用**OnlineOffline**模型，而不是其他的模型?



状态模型

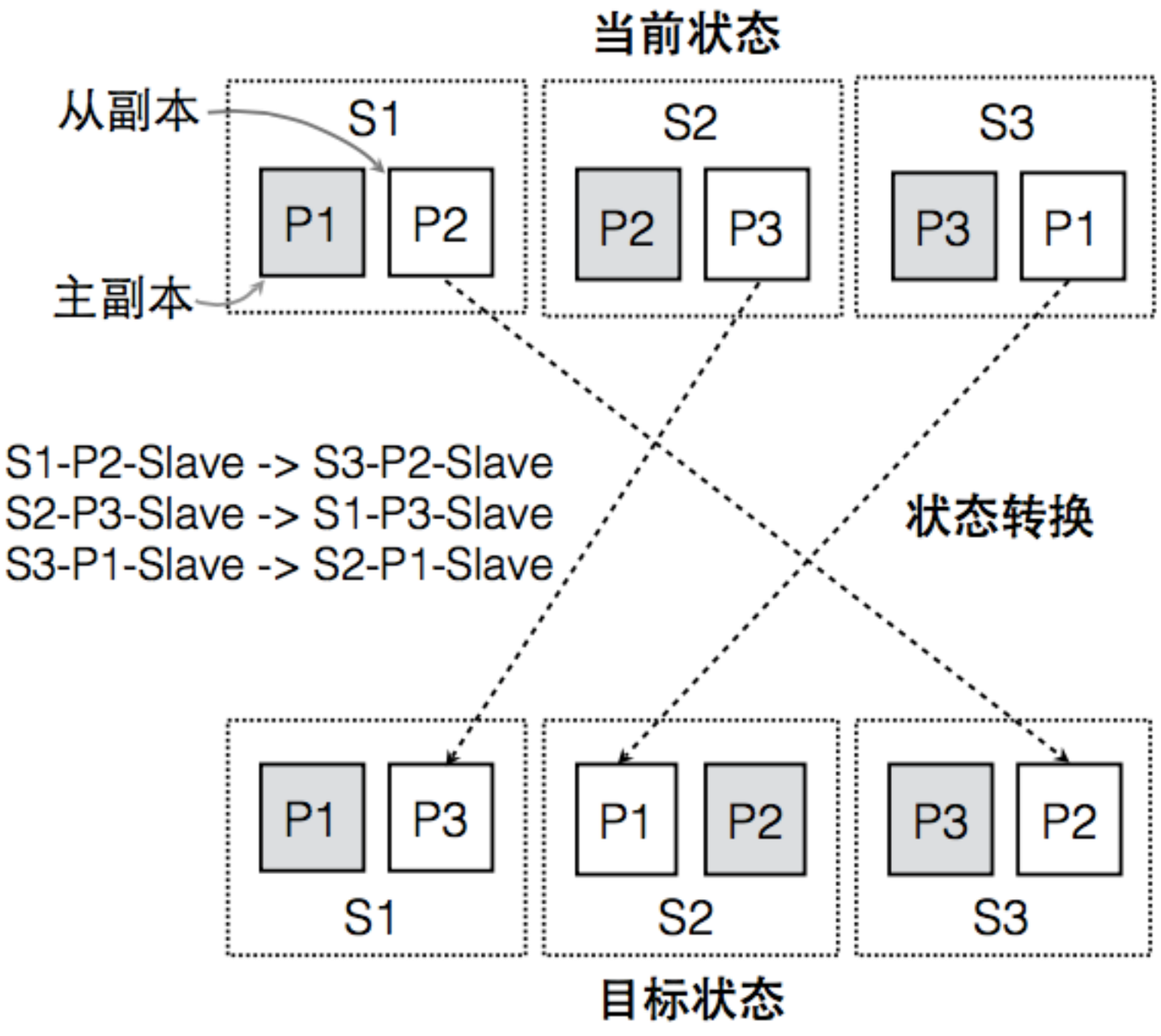
- MASTER-SLAVE
- **ONLINE-OFFLINE** ✅
- LEADER-STANDBY

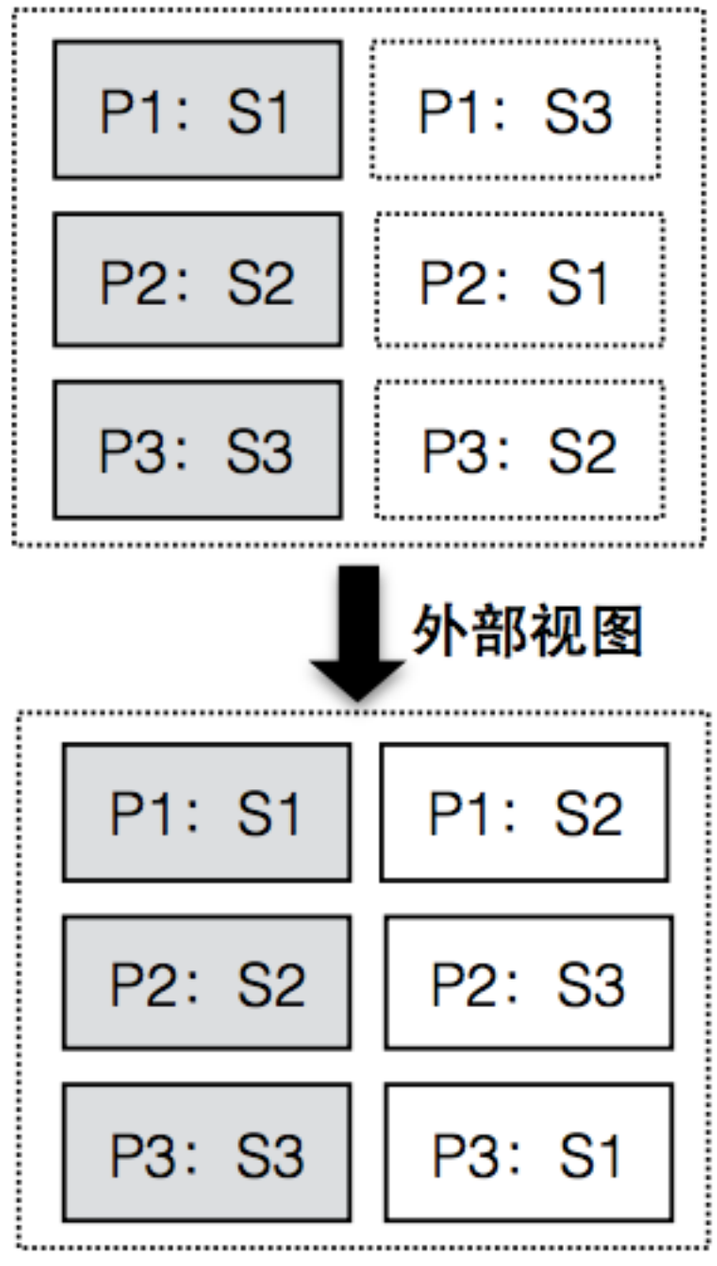
Kafka的术语	Helix的术语
主题（topic）	资源（resource）
分区（partition）	分区（partition）
MirrorMaker控制器（`HelixMirrorMakerManager`）	Helix控制器（`Controller`）
MirrorMaker工作节点（`MirrorMakerWorker`）	Helix参与者（`Participant`）

HELIX

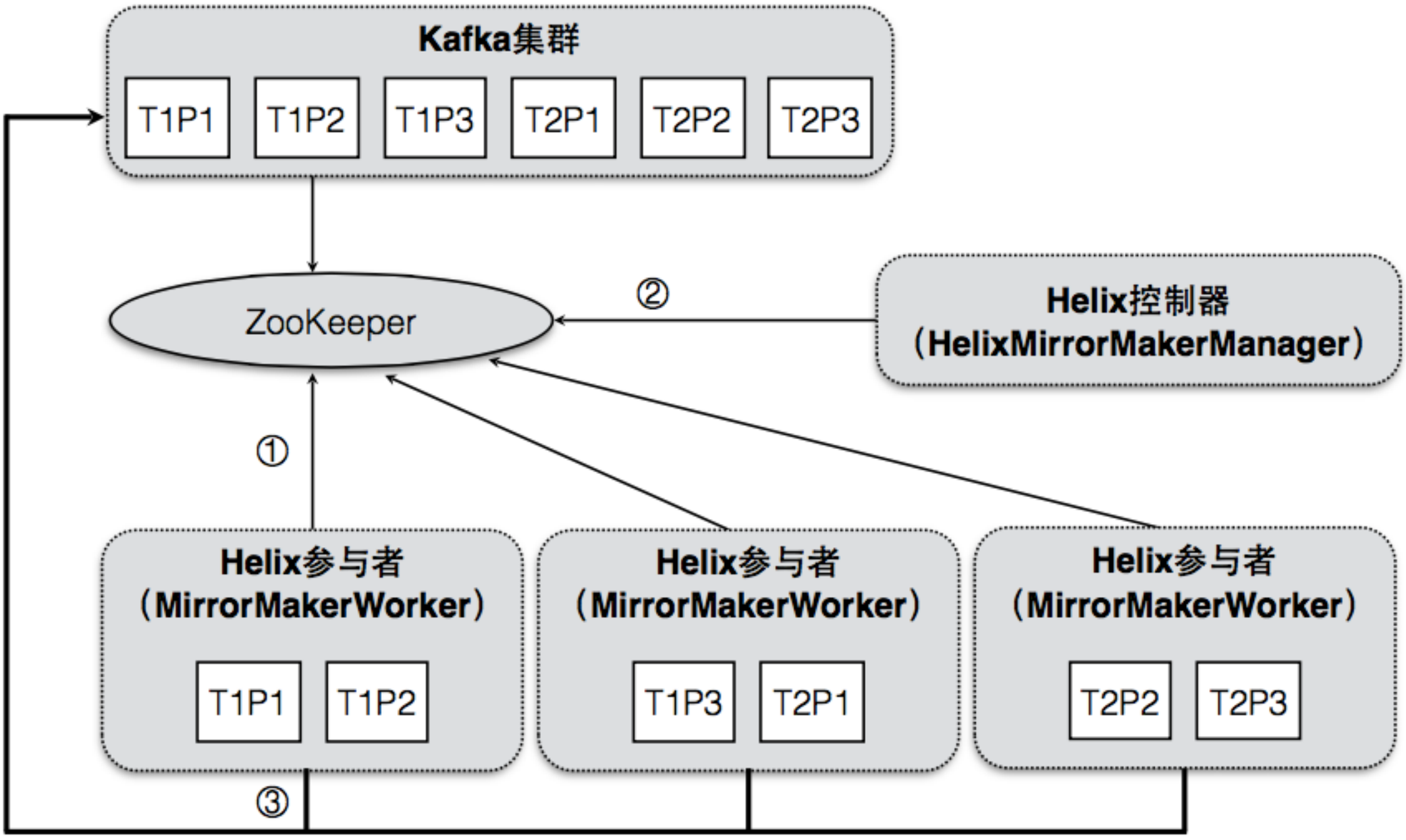
A cluster management framework for partitioned and replicated distributed resources

Helix状态机（Master Slave）

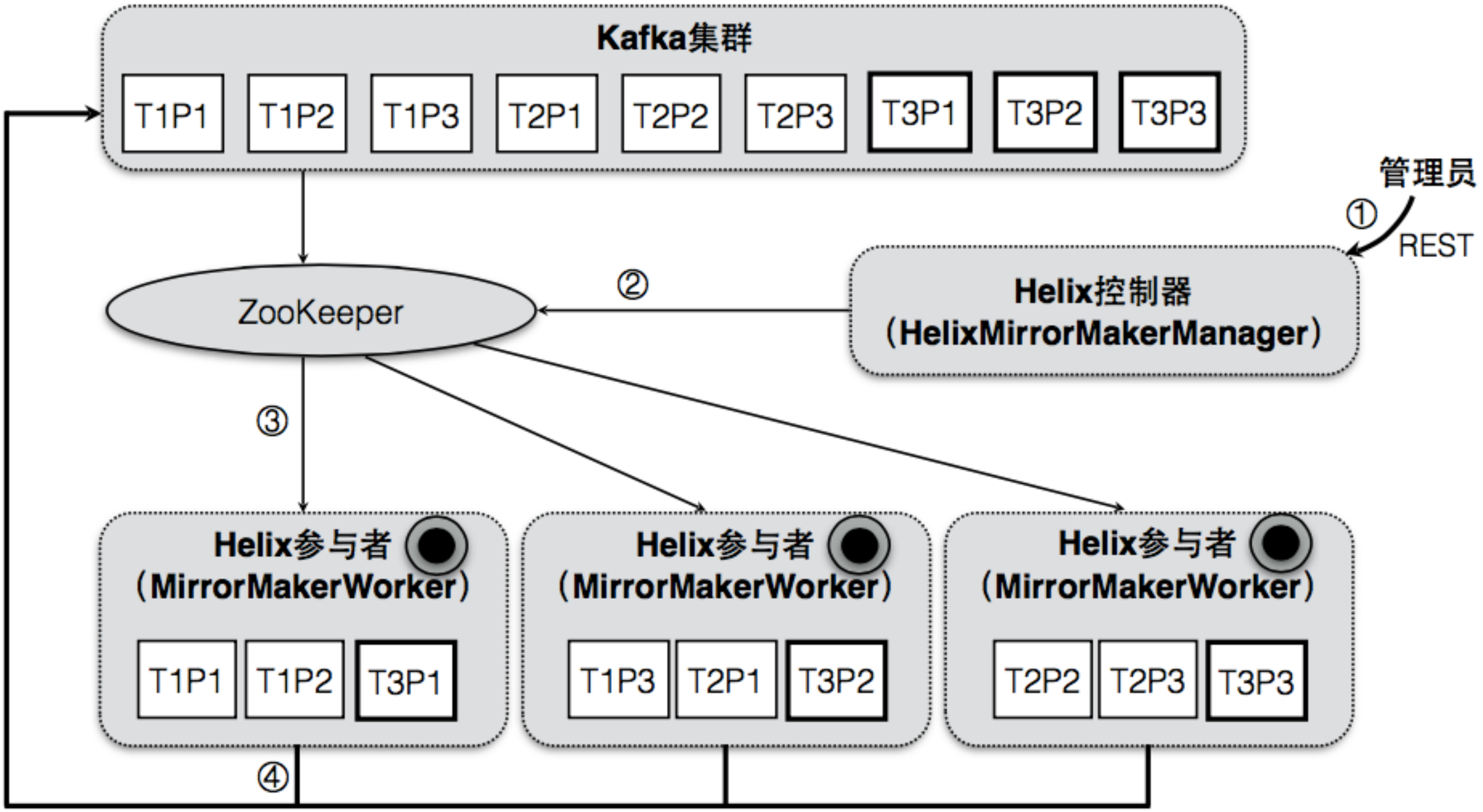




uReplicator工作原理

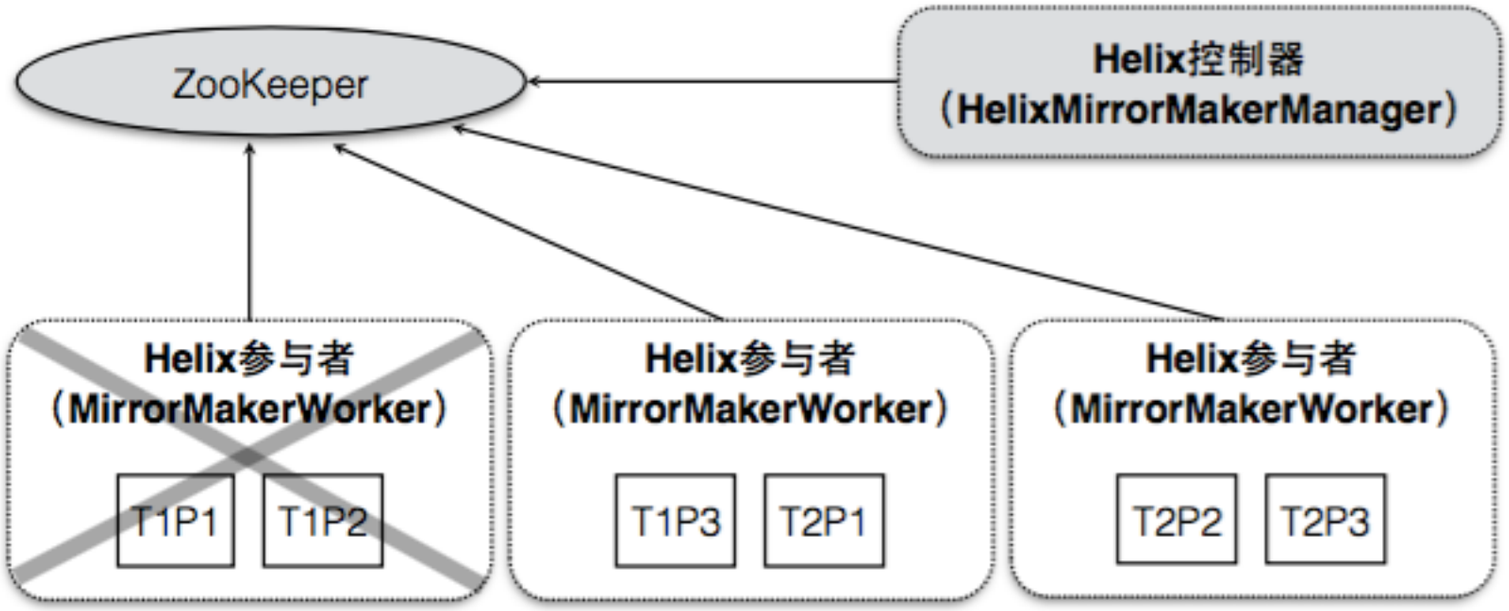


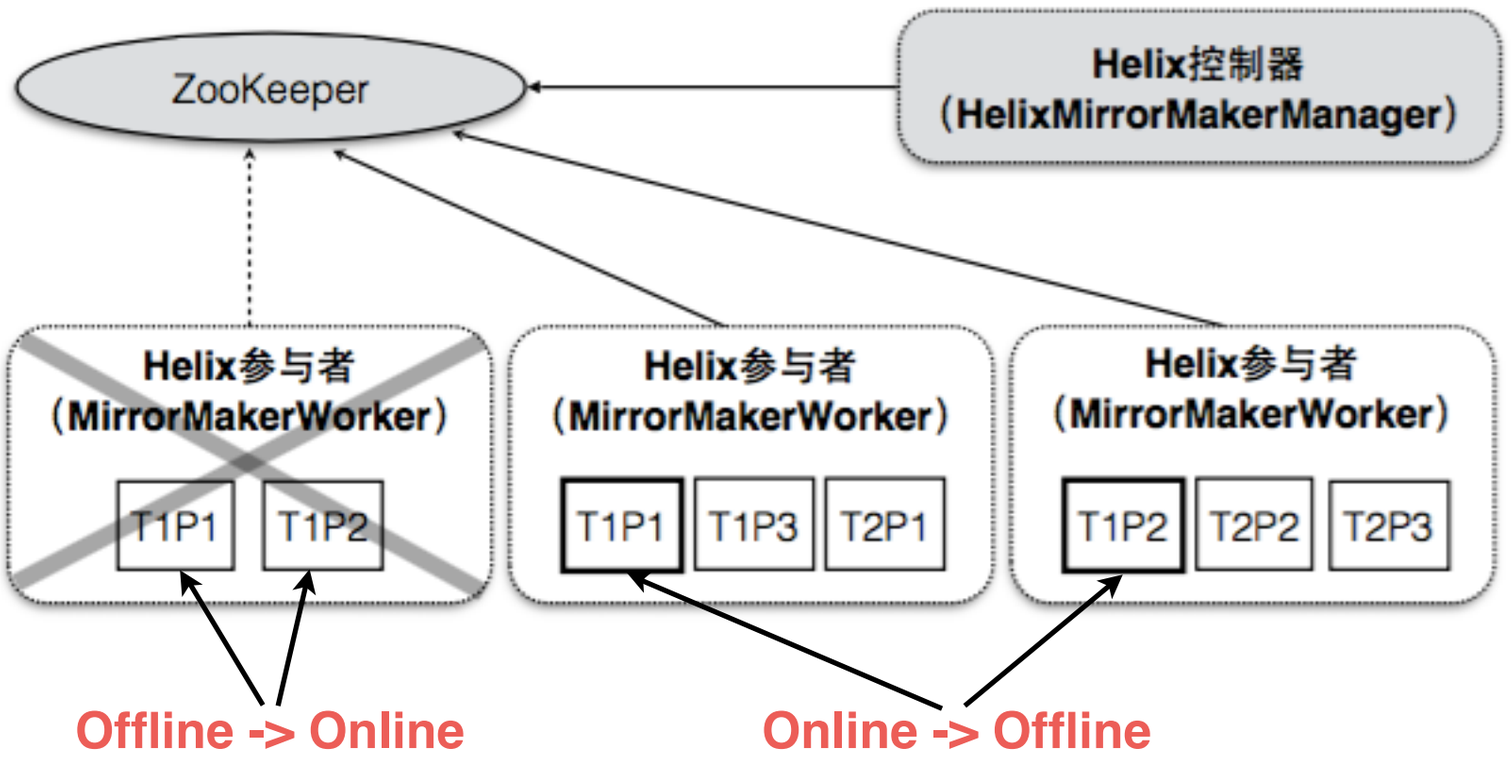
新增分区



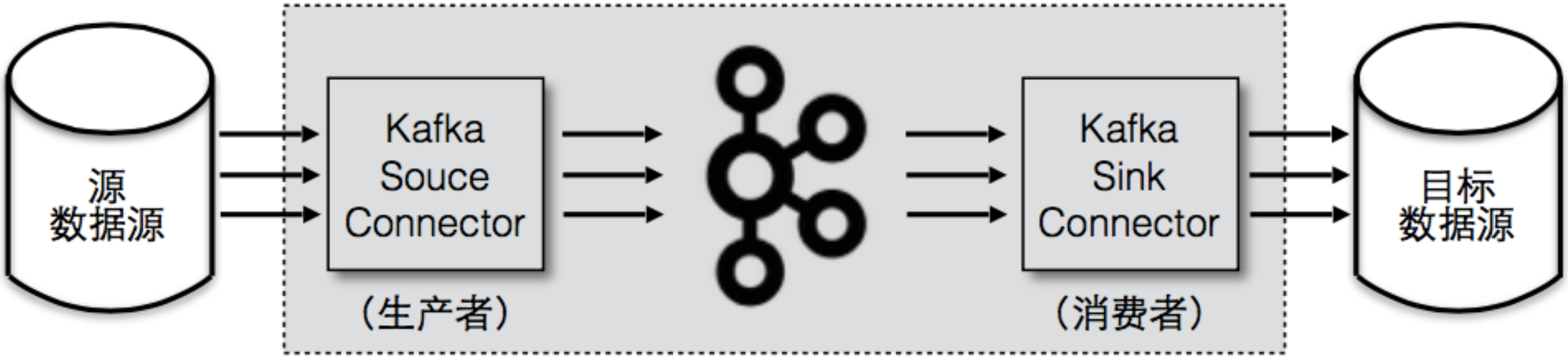
Offline -> Online

Worker节点挂掉


ZooKeeper
Helix控制器
（HelixMirrorMakerManager）
Helix参与者
（MirrorMakerWorker）
T1P1
T1P2
Helix参与者
（MirrorMakerWorker）
T1P3
T2P1
Helix参与者
（MirrorMakerWorker）
T2P2
T2P3


ZooKeeper
Helix控制器
（HelixMirrorMakerManager）
Helix参与者
（MirrorMakerWorker）
T1P1
T1P2
Helix参与者
（MirrorMakerWorker）
T1P1
T1P3
T2P1
Helix参与者
（MirrorMakerWorker）
T1P2
T2P2
T2P3
Offline -> Online
Online -> Offline

Kafka Connect



一个Kafka Connect进程封装的内部组件

```
$ cat config/connect-file-source.json
{
  "name": "connect-file-source",
  "config": {
    "connector.class":"FileStreamSourceConnector",
    "tasks.max":"1",
    "topic":"connect-test",
    "file":"test.txt"
  }
}
```

读取文件

```
$ cat config/connect-file-source2.json
{
  "name": "connect-file-source2",
  "config": {
    "connector.class":"FileStreamSourceConnector",
    "tasks.max":"1",
    "topic":"connect-test",
    "file":"test2.txt"
  }
}
```

写入文件

启动两个进程

```
$ bin/connect-distributed.sh config/connect-distributed.properties
$ bin/connect-distributed.sh config/connect-distributed2.properties
```

读取文件，写Kafka

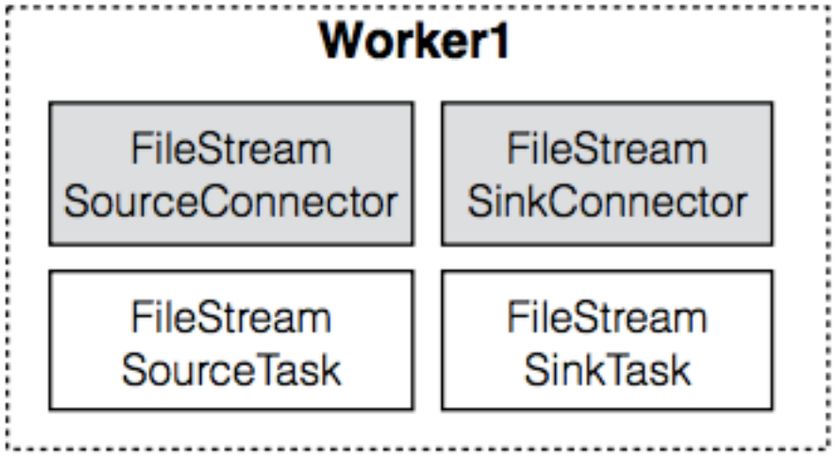
```
$ curl -X POST -H "Content-Type: application/json" -d \
@config/connect-file-source.json http://localhost:8083/connectors
```

读Kafka，写入文件

```
$ curl -X POST -H "Content-Type: application/json" -d \
@config/connect-file-source2.json http://localhost:8084/connectors
```

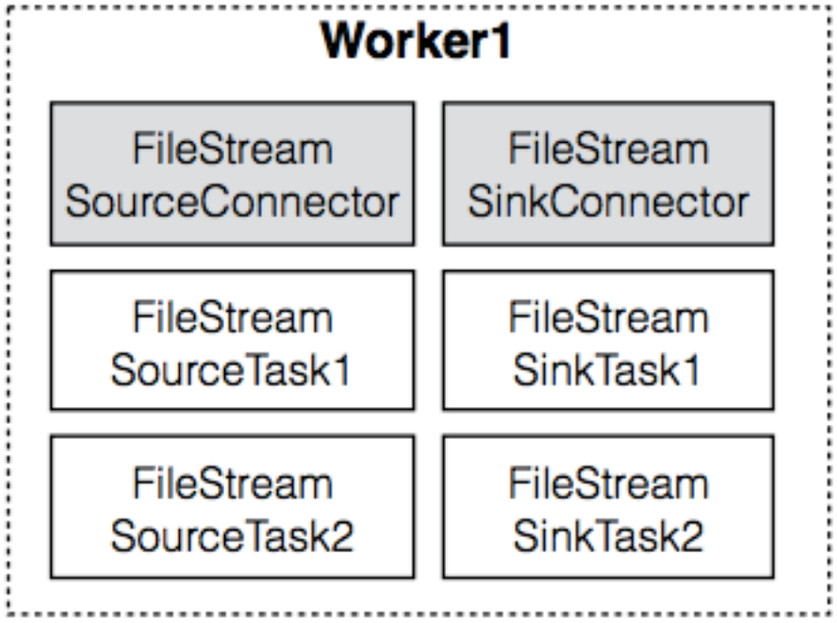
Kafka Connect的水平扩容

①



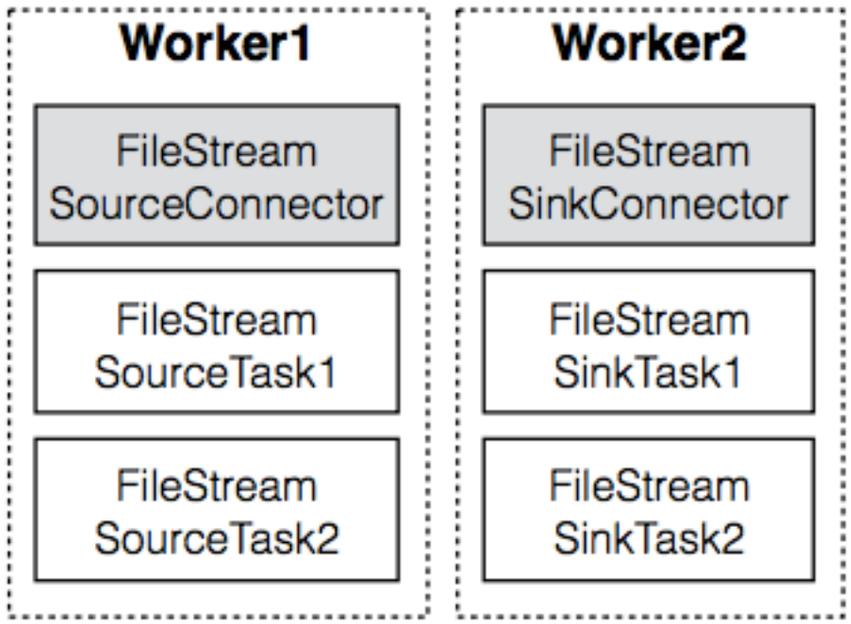
一个进程
一个任务

②



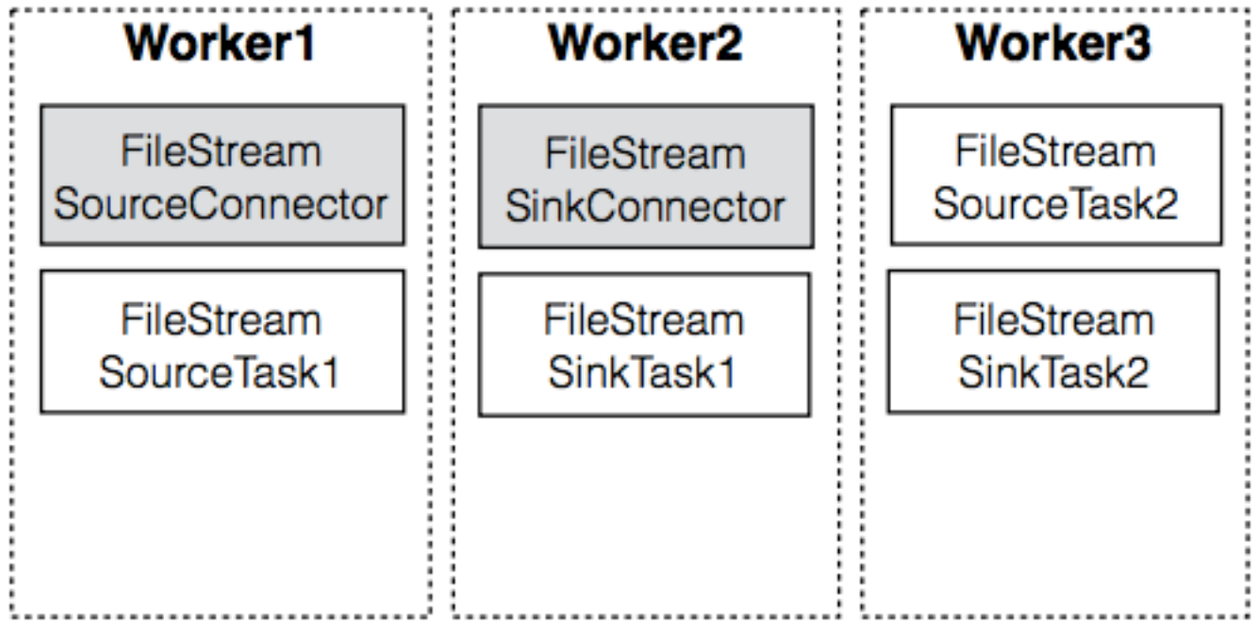
一个进程
多个任务

③

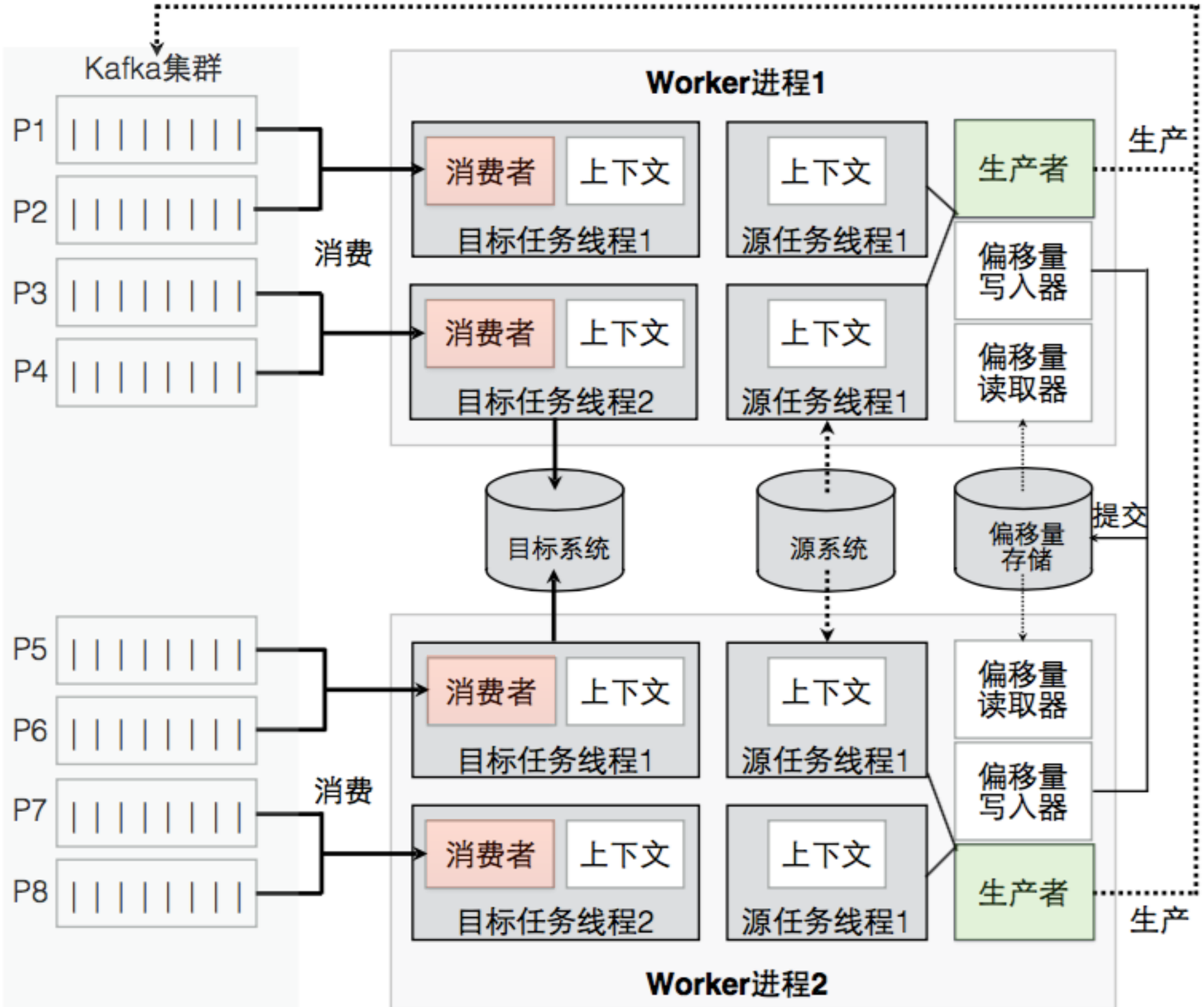


两个进程
多个任务

④



三个进程
多个任务


Kafka集群
P1
P2
P3
P4
消费
Worker进程1
消费者
上下文
目标任务线程1
上下文
源任务线程1
生产者
生产
偏移量
写入器
消费者
上下文
目标任务线程2
上下文
源任务线程1
偏移量
读取器
目标系统
源系统
偏移量
存储
提交
P5
P6
P7
P8
消费
消费者
上下文
目标任务线程1
上下文
源任务线程1
偏移量
读取器
偏移量
写入器
消费者
上下文
目标任务线程2
上下文
源任务线程1
生产者
生产
Worker进程2

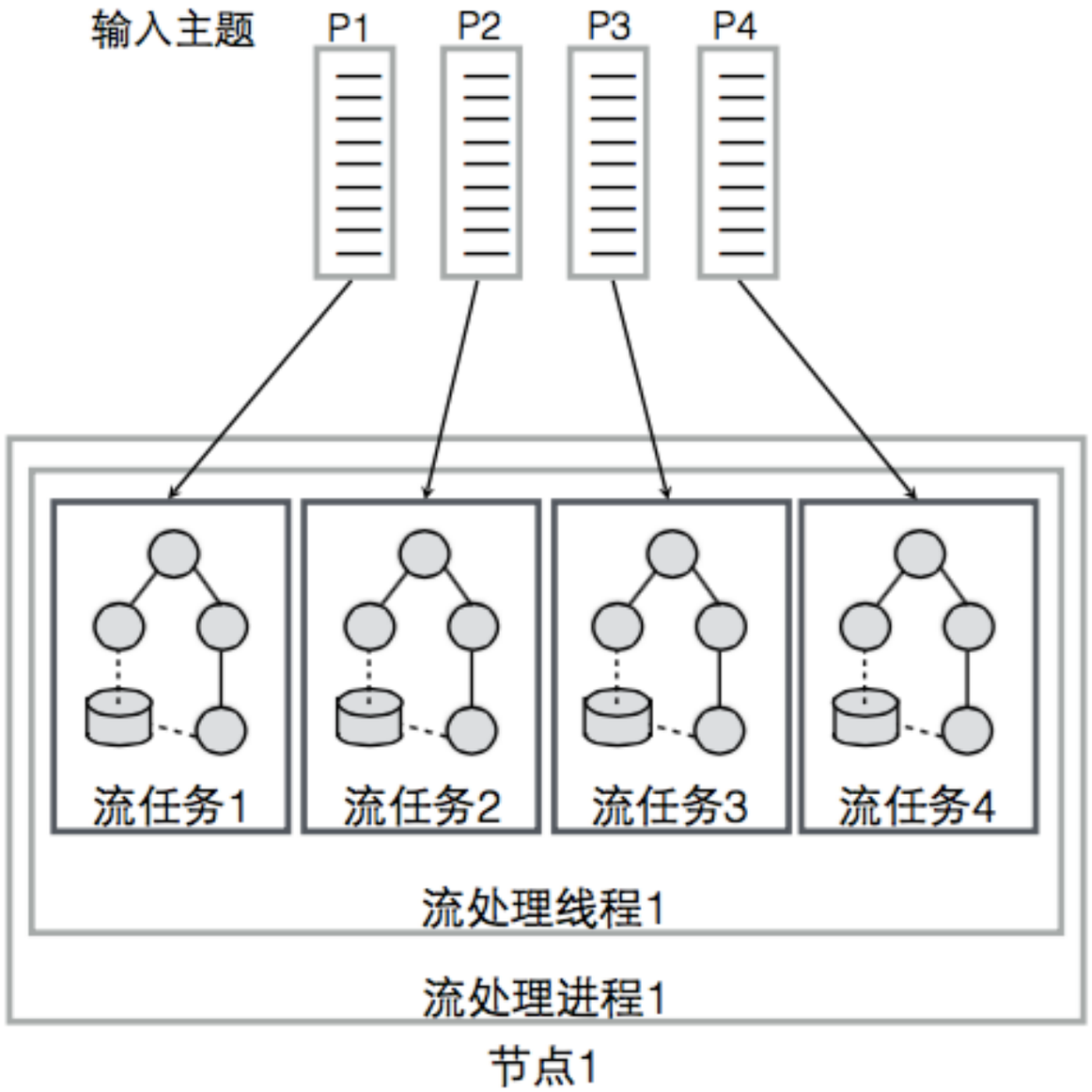
Stream Processing has some hard parts

- ordering 日志有序
- state 本地状态存储 变更日志主题
- partitioning & scalability 分区
- reprocessing 重置偏移量
- fault tolerance 组协调协议，备份任务
- time, windowing & out-of-order data 窗口状态存储

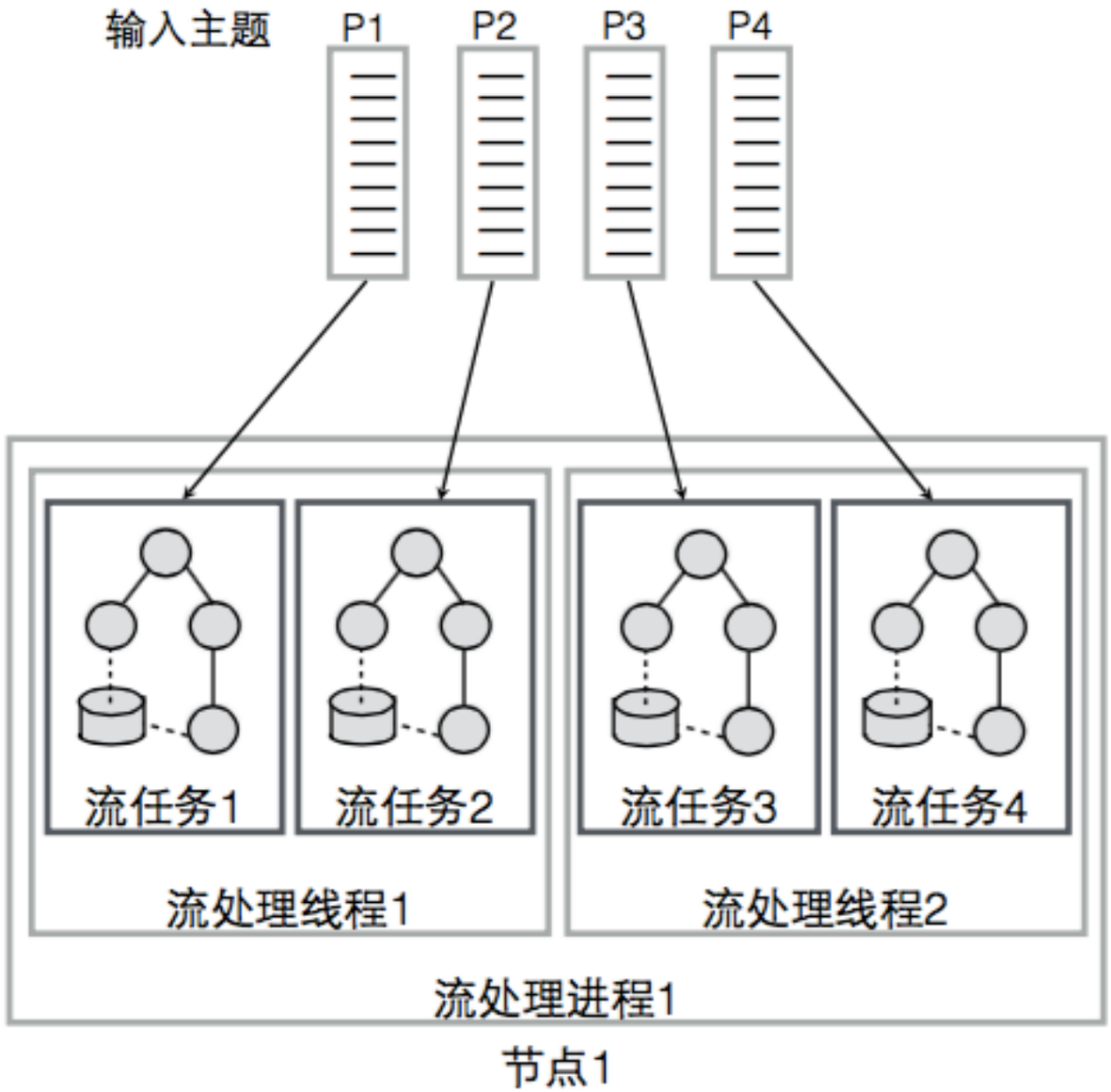
https://qconnewyork.com/ny2016/ny2016/presentation/large-scale-stream-processing-apache-kafka.html

Kafka Streams的线程模型

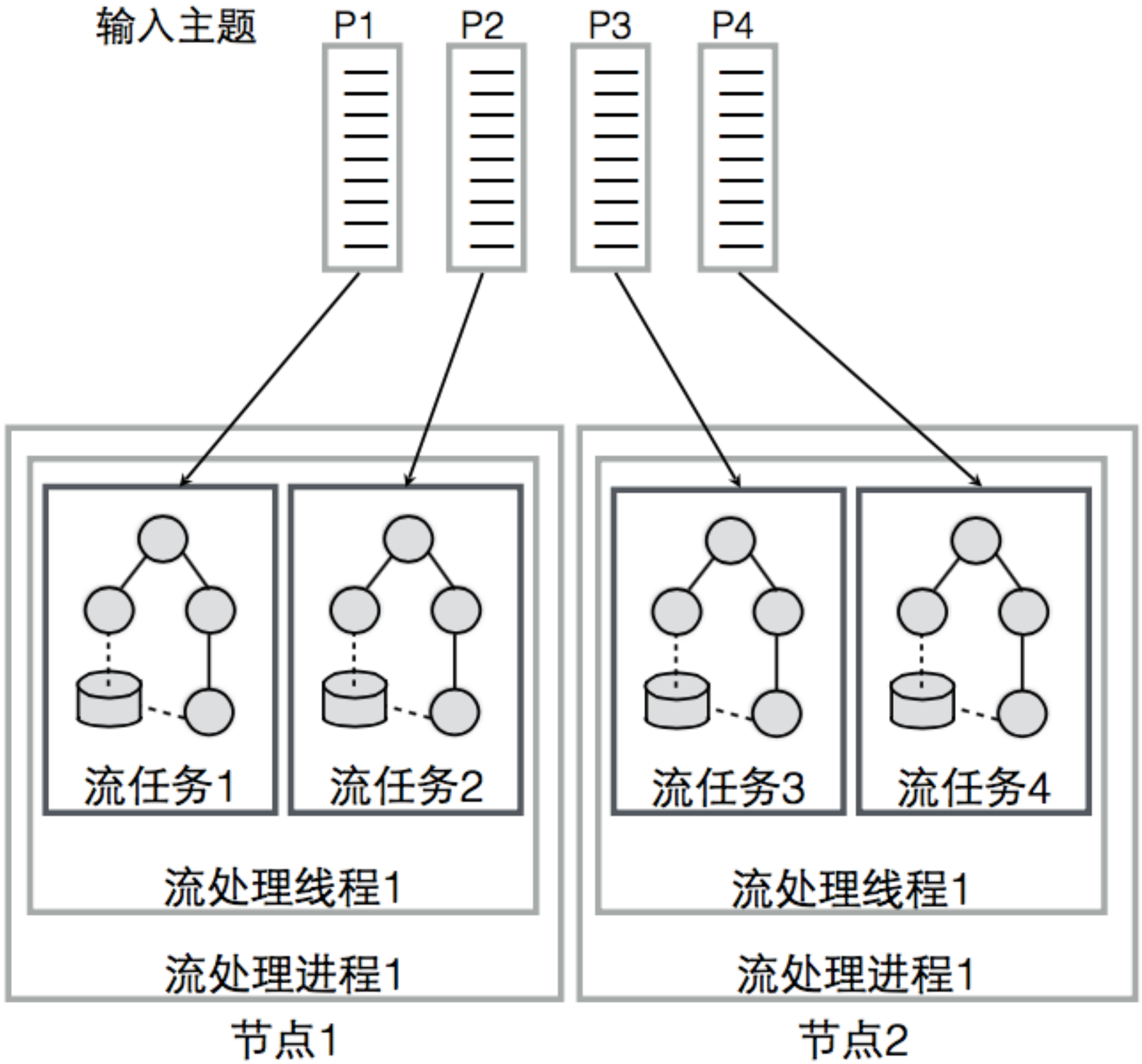
任务分配机制还是使用了“组管理协议”，但是相对来说会复杂点



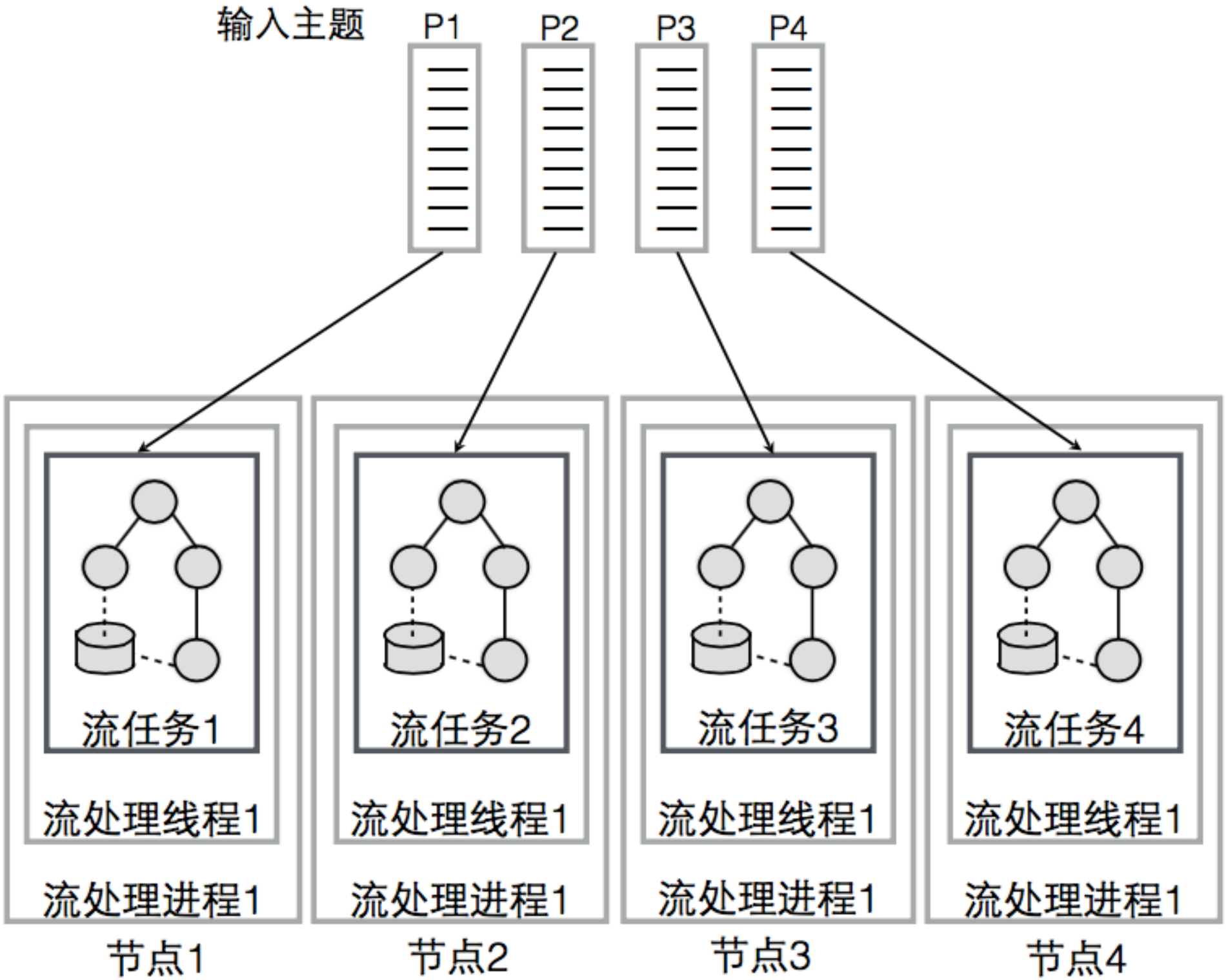
一个节点
一个进程
一个线程



一个节点
一个进程
多个线程

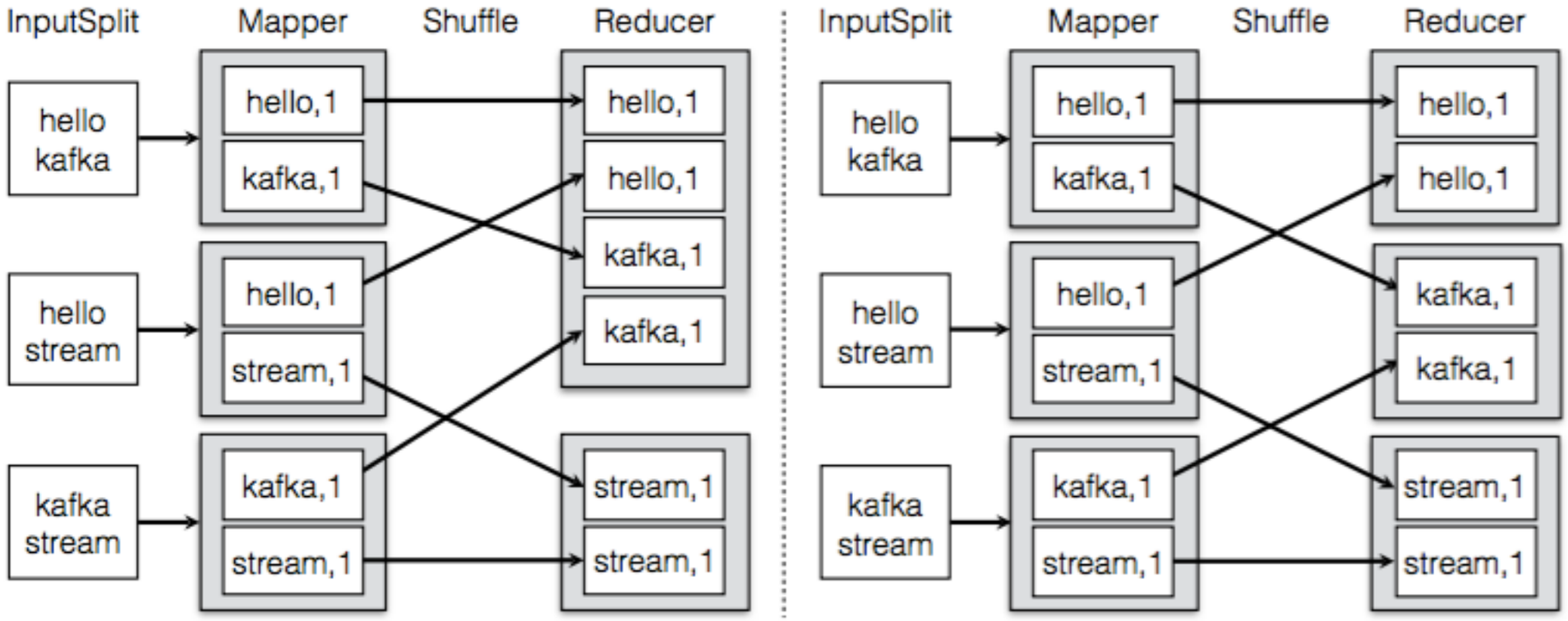

输入主题
P1
P2
P3
P4
流任务1
流任务2
流任务3
流任务4
流处理线程1
流处理线程1
流处理进程1
流处理进程1
节点1
节点2

两个节点


输入主题
P1
P2
P3
P4
流任务1
流任务2
流任务3
流任务4
流处理线程1
流处理线程1
流处理线程1
流处理线程1
流处理进程1
流处理进程1
流处理进程1
流处理进程1
节点1
节点2
节点3
节点4

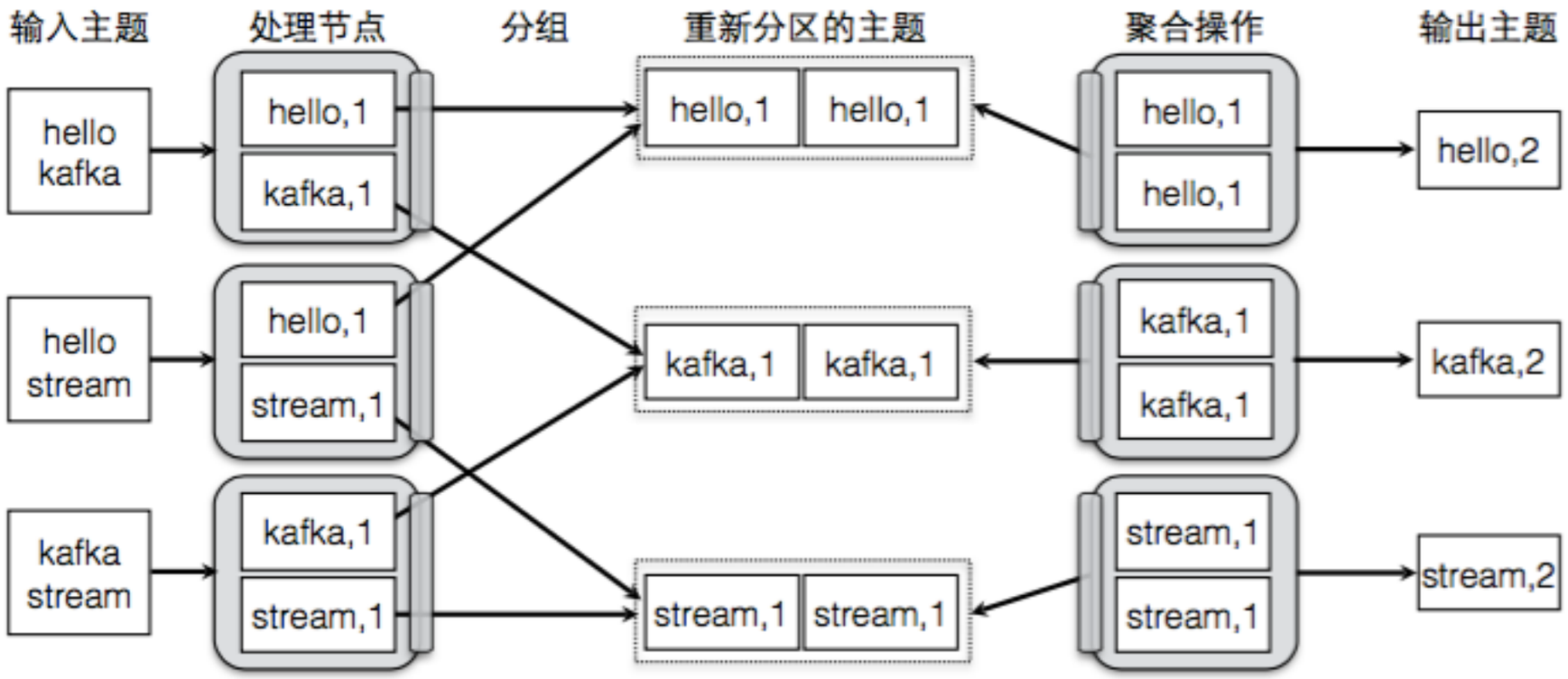
两个节点

Kafka流处理的“Shuffle”

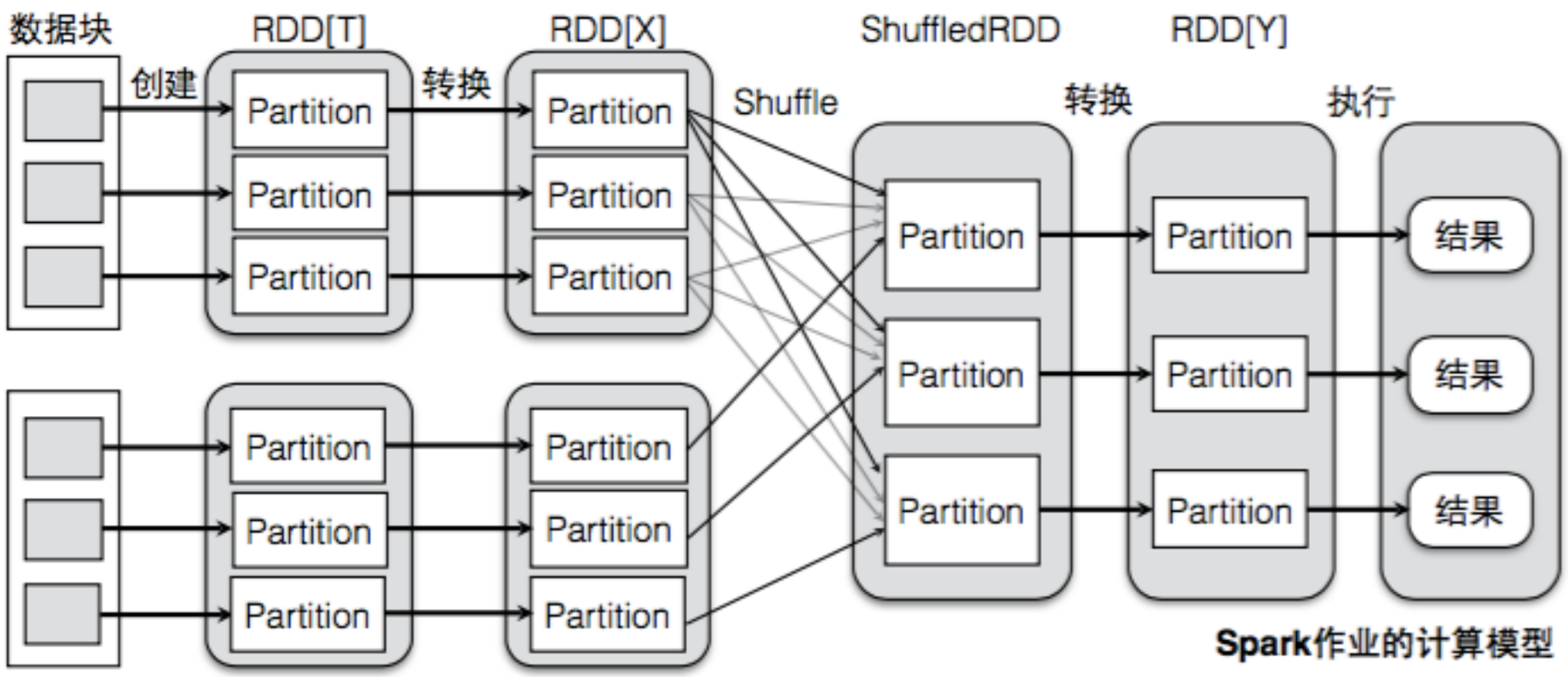


Hadoop的MapReduce模型

Kafka的重新分区模型

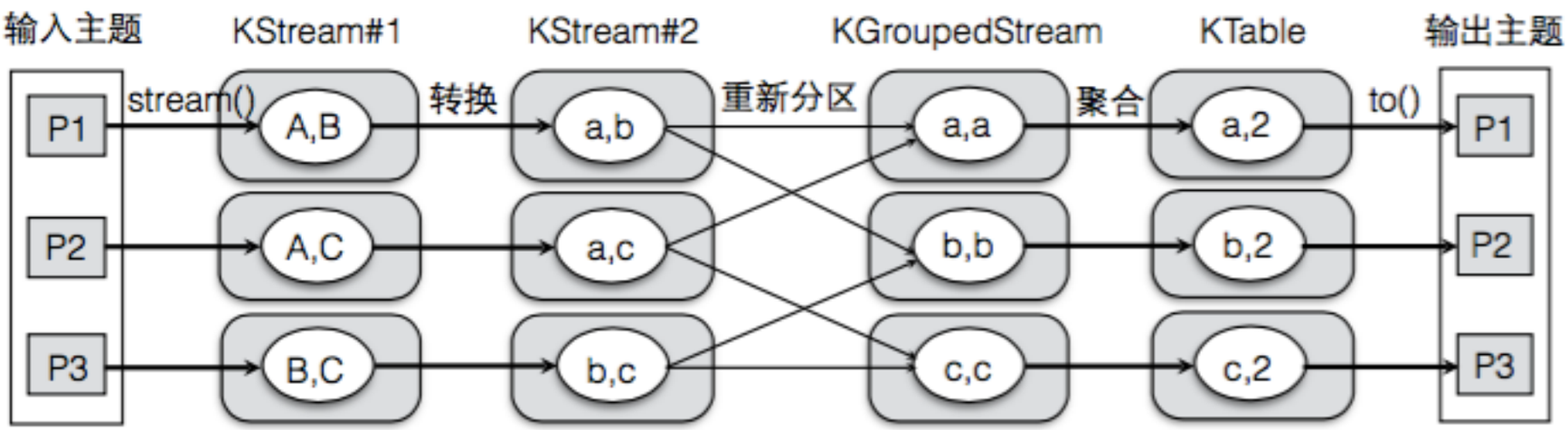


Kafka流处理的计算模型



Spark作业的计算模型

Kafka流处理的计算模型



Storm

```java
public static void main(String[] args) throws Exception {
    TopologyBuilder builder = new TopologyBuilder();
    builder.setSpout("spout", new RandomSentenceSpout(), 1);

    builder.setBolt("split", new SplitSentenceBolt(), 2)
      .shuffleGrouping("spout");
    builder.setBolt("count", new WordCountBolt(), 2)
      .fieldsGrouping("split", new Fields("word"));
    builder.setBolt("print", new PrinterBolt(), 1)
      .shuffleGrouping("count");

    Topology topology = builder.createTopology()
    Config conf = new Config();
    conf.setDebug(false);
    if (args != null && args.length > 0) {
        conf.setNumWorkers(3);
        StormSubmitter.submitTopology(args[0], conf, topology);
    } else {
        conf.setMaxTaskParallelism(3);
        LocalCluster cluster = new LocalCluster();
        cluster.submitTopology("word-count", conf, topology);
        Thread.sleep(10000);
        cluster.shutdown();
    }
}
```

```java
TridentTopology topology = new TridentTopology();
TridentState wordCounts = topology.newStream("spout1", spout)
    .each(new Fields("sentence"),new Split(),new Fields("word"))
    .groupBy(new Fields("word"))
    .persistentAggregate(new MemoryMapState.Factory(),
        new Count(), new Fields("count"))
    .parallelismHint(6);
```

Kafka Streams

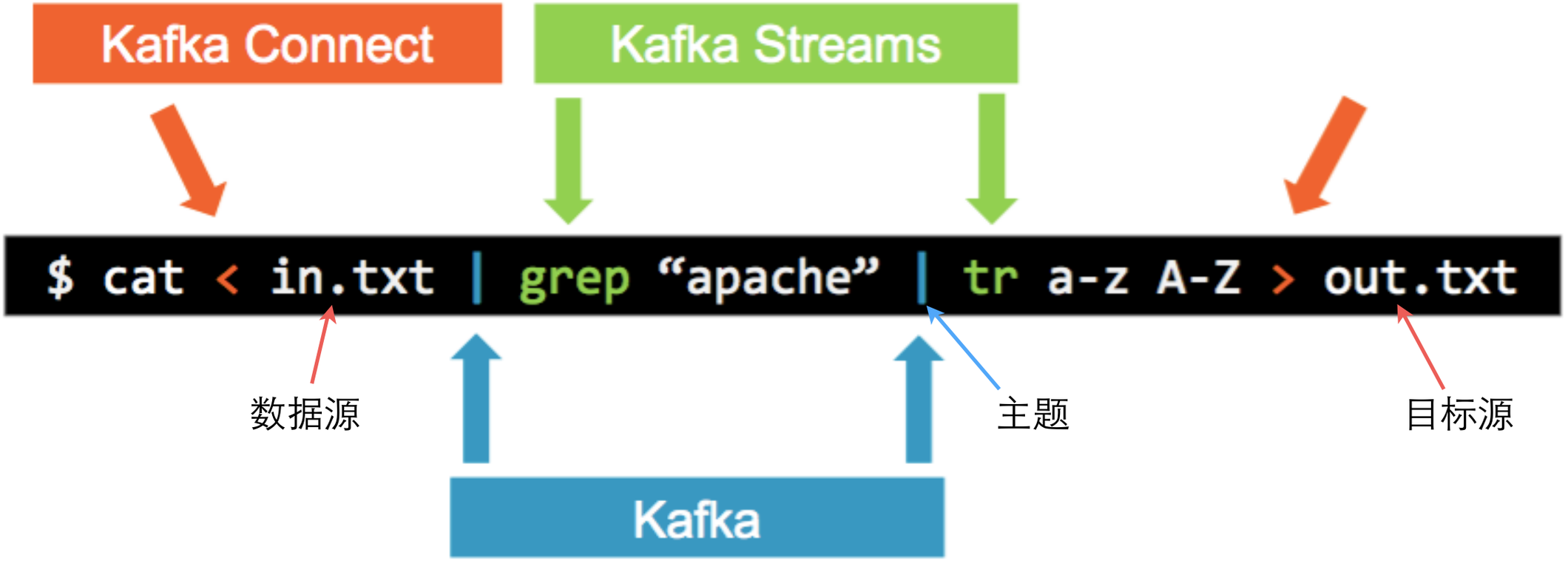
```java
public static void main(String[] args) throws Exception {
  // Kafka流处理应用程序的配置
  Properties props = new Properties();
  props.put(APPLICATION_ID_CONFIG, "streams-wordcount");
  props.put(BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
  props.put(ZOOKEEPER_CONNECT_CONFIG, "localhost:2181");
  props.put(KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
  props.put(VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());
  props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

  // 构建Kafka流处理的拓扑
  TopologyBuilder builder = new TopologyBuilder();
  builder.addSource("Source", "streams-input1");
  builder.addProcessor("Process", new MyProcessorSupplier(), "Source");
  builder.addStateStore(Stores.create("Counts")
    .withStringKeys().withIntegerValues().inMemory().build(), "Process");
  builder.addSink("Sink", "streams-output1", "Process");

  KafkaStreams streams = new KafkaStreams(builder, props); // 创建实例
  streams.start(); // 启动KafkaStreams实例
}
```

```java
public static void main(String[] args) throws Exception {
  Properties props = new Properties();
  props.put(APPLICATION_ID_CONFIG, "dsl-wc");
  props.put(BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
  // 使用高级DSL方式构建拓扑
  KStreamBuilder builder = new KStreamBuilder();
  KStream<String, String> stream = builder.stream("dsl-input1");
  KTable<String, Long> countTable = stream
    .flatMapValues(value -> Arrays.asList(value.split(" ")))
    .groupBy((key, word) -> word).count("Counts1");
    .to("ktable-output1");

  KafkaStreams streams = new KafkaStreams(builder, props);
  streams.start();
}
```


Kafka Connect
Kafka Streams
$ cat < in.txt | grep “apache” | tr a-z A-Z > out.txt
数据源
主题
目标源
Kafka

更多资源

https://www.confluent.io/blog/

https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Improvement+Proposals

http://www.jasongj.com/tags/Kafka/

http://zqhxuyuan.github.io/tags/kafka/

http://www.cnblogs.com/huxi2b/

https://www.iteblog.com/archives/category/kafka/

https://github.com/oldratlee/translations/tree/master/log-what-every-software-engineer-should-know-about-real-time-datas-unifying

Tks…