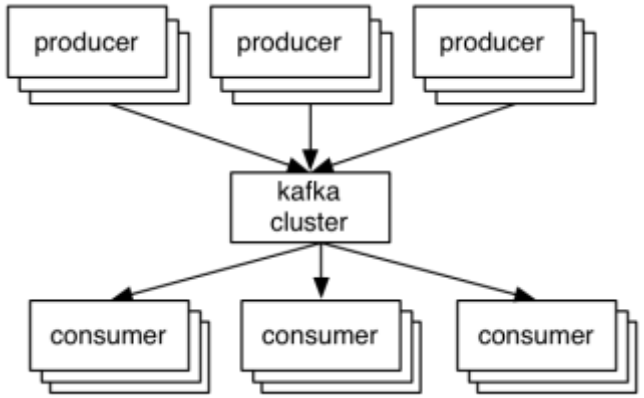
Kafka Core Concepts

Kafka core concepts

- **Records** have a **key (optional), value**, and **timestamp**.
- **Topic** a stream of records (e.g., orders), feed name
  - **Log** topic storage on disk
  - **Partition** / segments (parts of topic log)
- **Producer** API to produce streams of records
- **Consumer** API to consume streams of records
- **Broker** Kafka server that runs in a Kafka Cluster. Brokers form a cluster. Acluster consists of many Kafka Brokers on many servers.
- **ZooKeeper** Does coordination of brokers/cluster topology. Consistent file system for configuration information and leadership election for Broker Topic Partition Leaders.

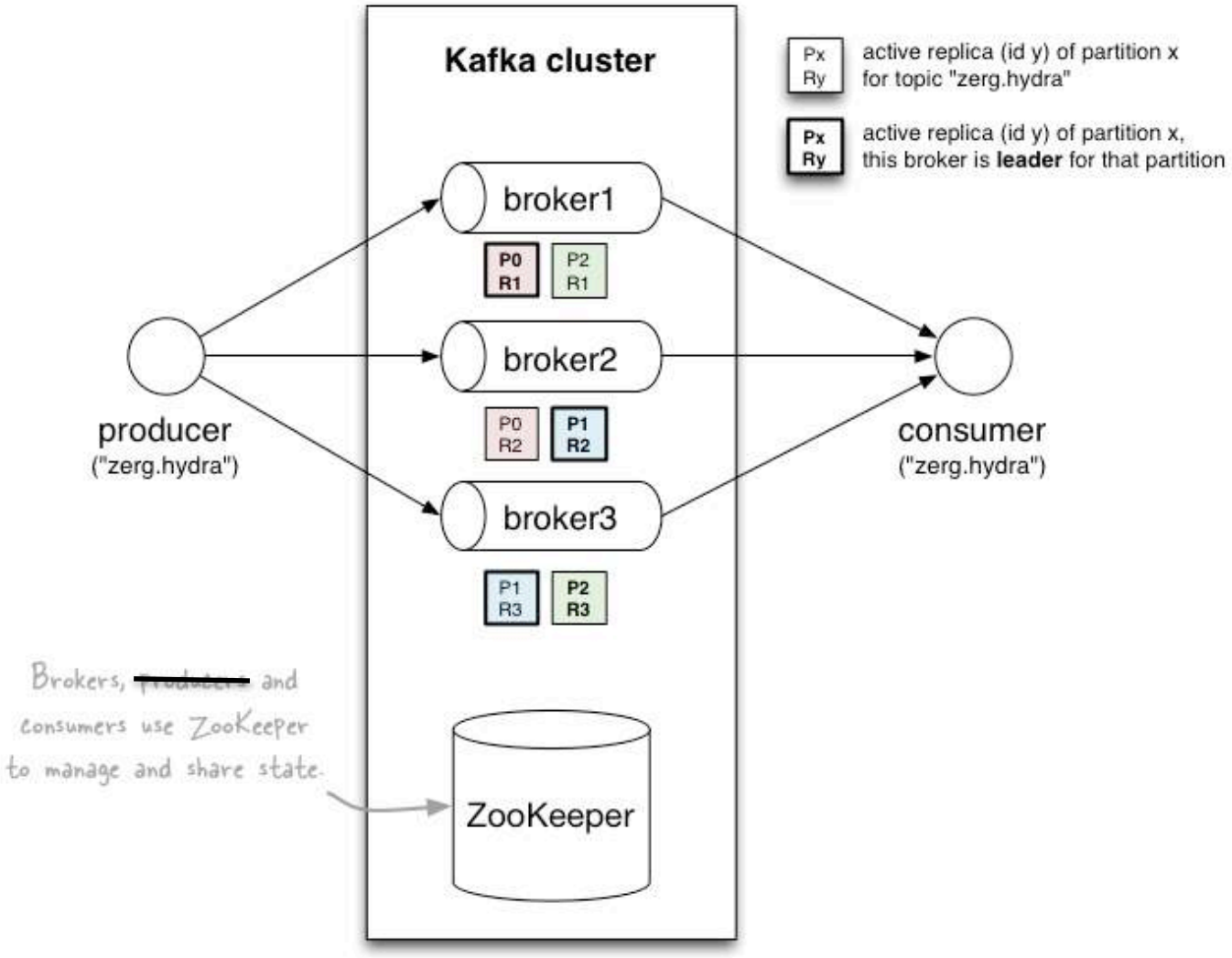
A first look

- The who is who
  - **Producers** write data to **brokers**.
  - **Consumers** read data from **brokers**.
  - All this is distributed.



- The data
  - Data is stored in **topics**.
  - **Topics** are split into **partitions**, which are **replicated**.

A first look

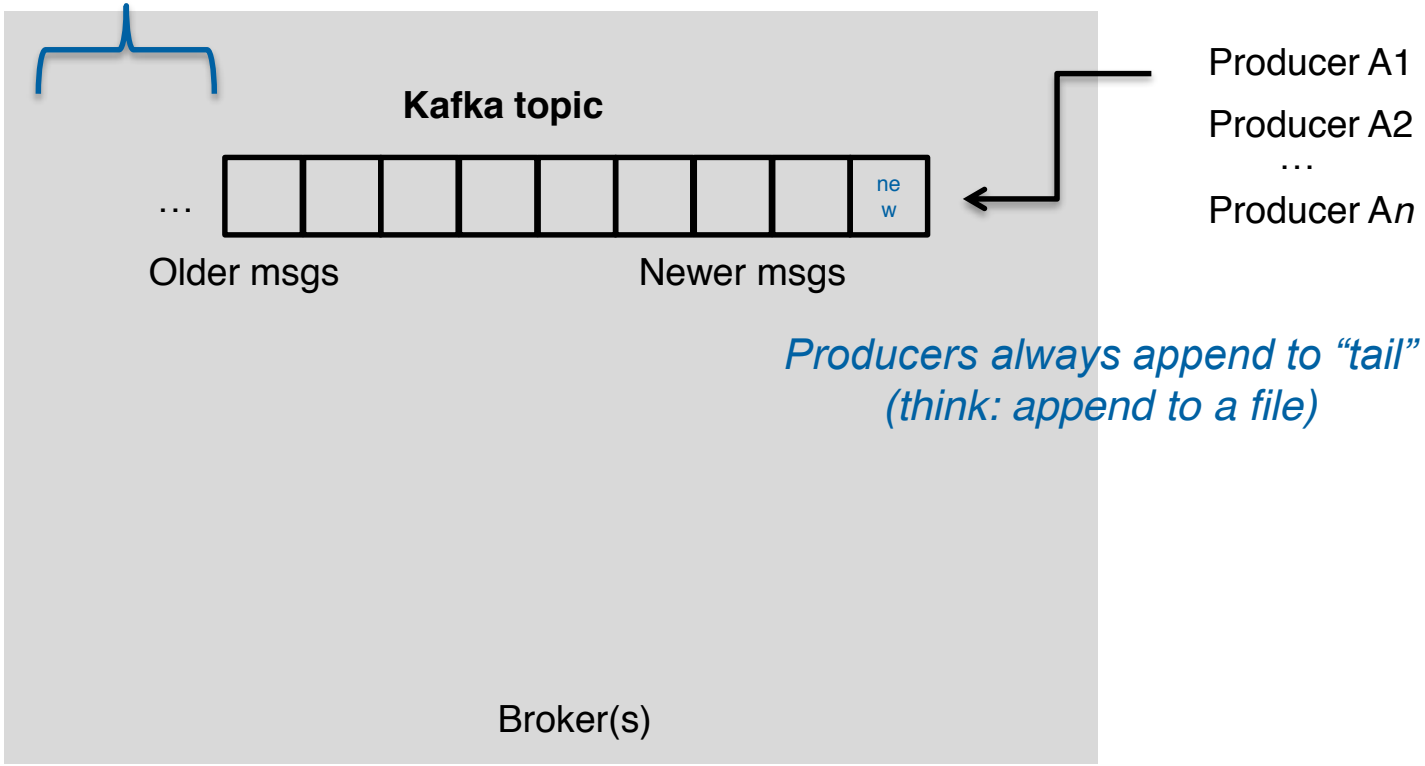


http://www.michael-noll.com/blog/2013/03/13/running-a-multi-broker-apache-kafka-cluster-on-a-single-node/

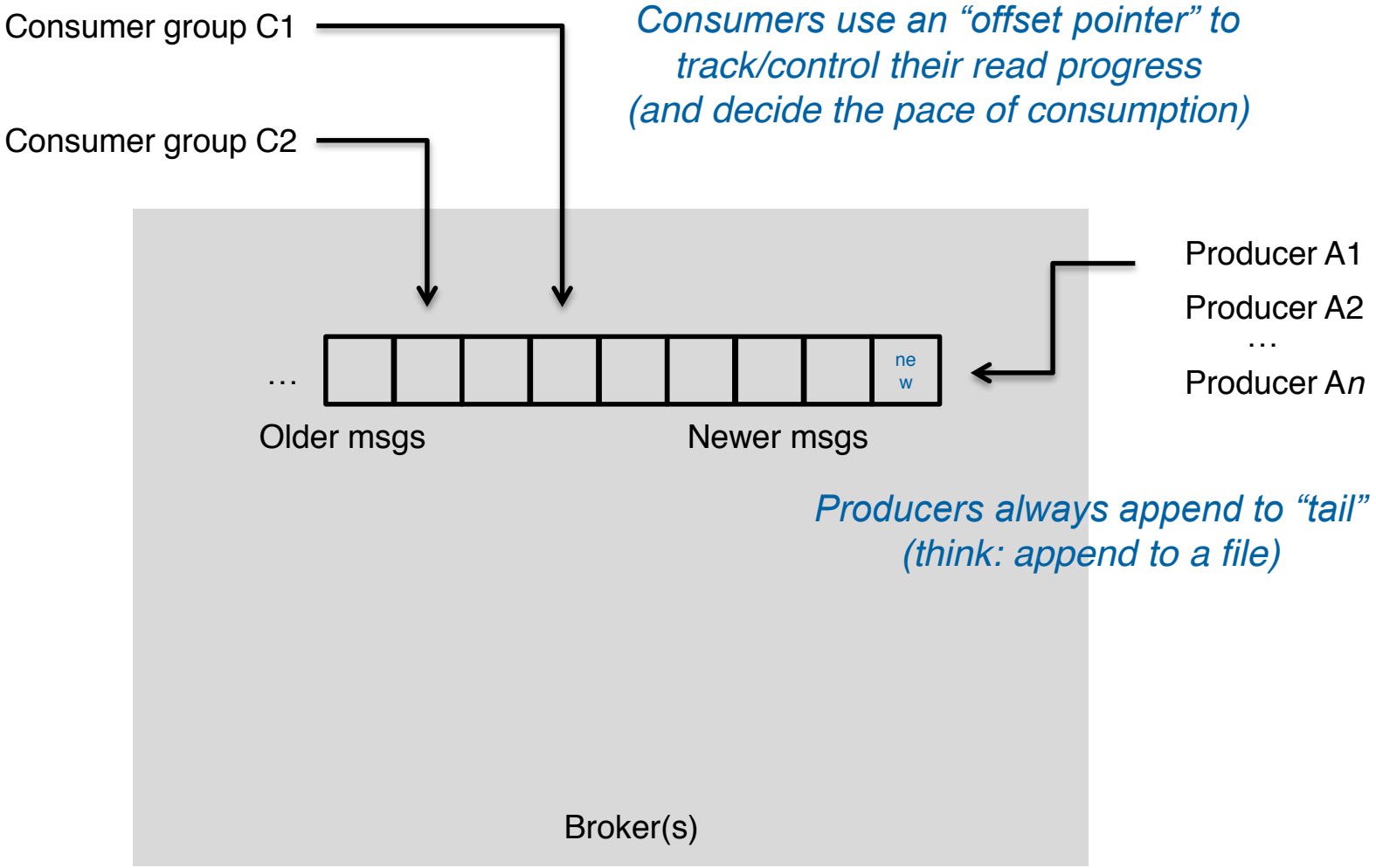
Topics

- **Topic:** feed name to which messages are published
  - Example: “zerg.hydra”

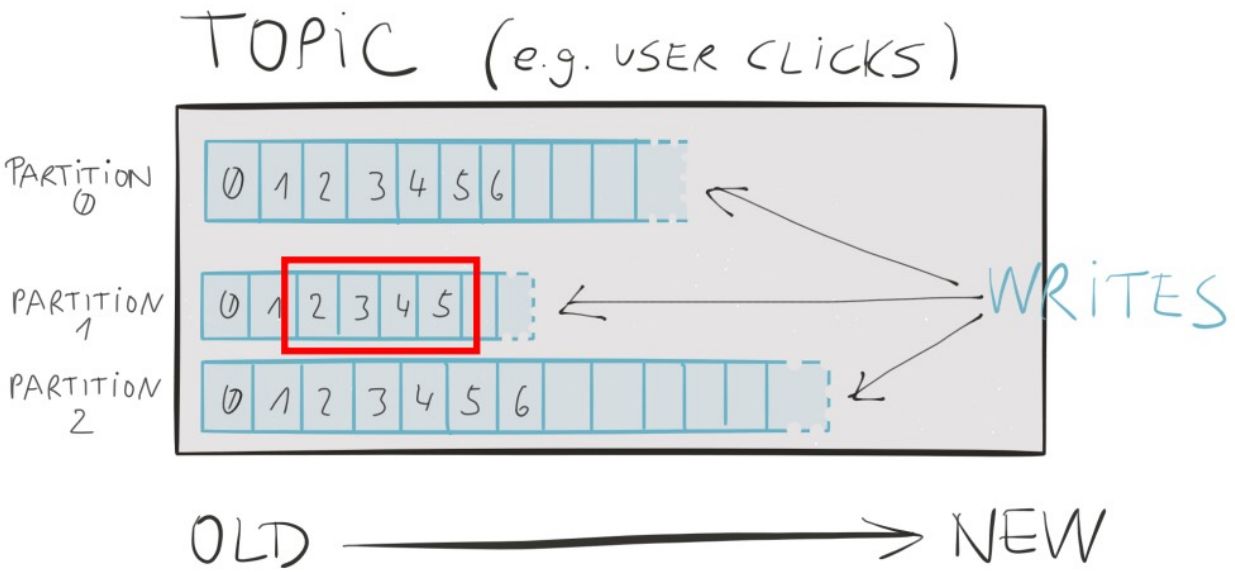
Kafka prunes “head” based on age or max size or “key”

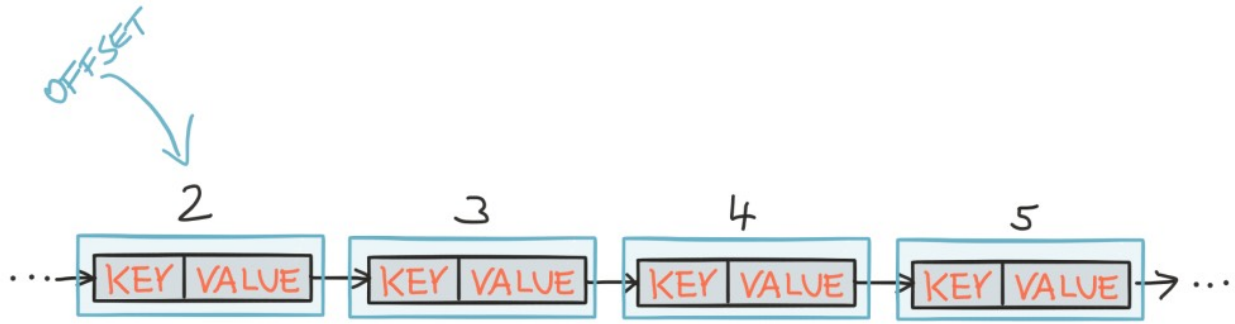


Topics


Consumer group C1
Consumer group C2
Consumers use an "offset pointer" to track/control their read progress (and decide the pace of consumption)
Producer A1
Producer A2
…
Producer An
…
new
Older msgs
Newer msgs
Producers always append to "tail" (think: append to a file)
Broker(s)

Topics





http://www.michael-noll.com/blog/2013/03/13/running-a-multi-broker-apache-kafka-cluster-on-a-single-node/

Topics

- Creating a topic
  - CLI

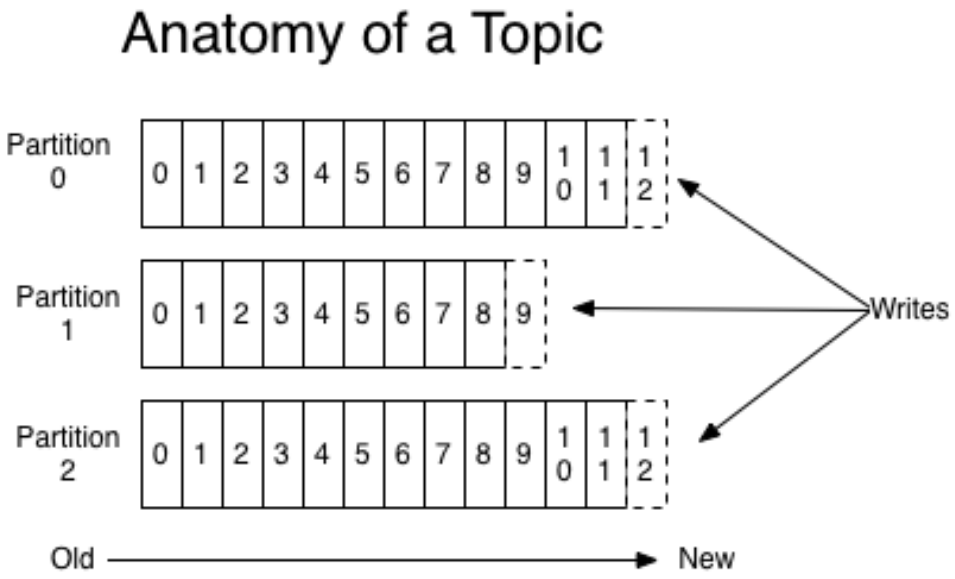
```
$ kafka-topics.sh --zookeeper zookeeper1:2181 --create --topic zerg.hydra \
    --partitions 3 --replication-factor 2 \
    --config x=y
```

  - API
    https://github.com/miguno/kafka-storm-starter/blob/develop/src/main/scala/com/miguno/kafkastorm/storm/KafkaStormDemo.scala
  - Auto-create via `auto.create.topics.enable = true`

- Modifying a topic
  - https://kafka.apache.org/documentation.html#basic_ops_modify_topic

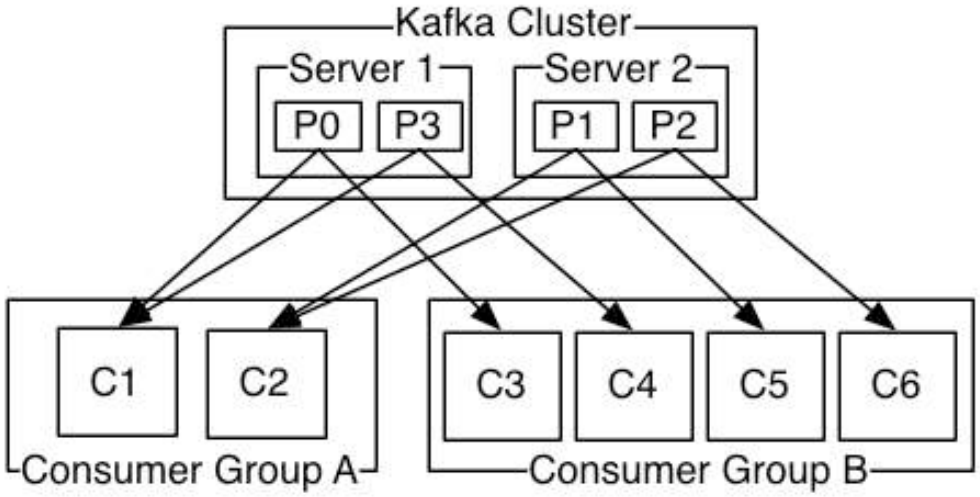
Partitions

- A topic consists of **partitions.**
- Partition: **ordered + immutable** sequence of messages that is continually appended to



Partitions

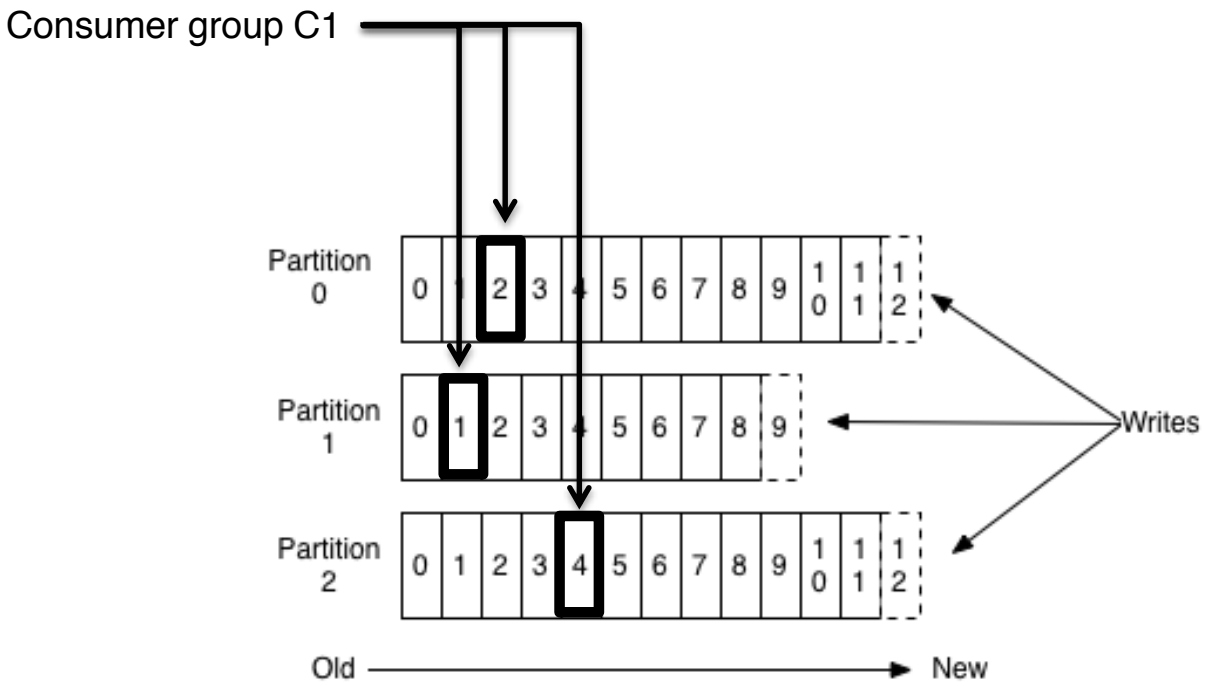
- #partitions of a topic is configurable
- #partitions determines **max** consumer (group) parallelism
  - Cf. parallelism of Storm's KafkaSpout via `builder.setSpout(,,N)`



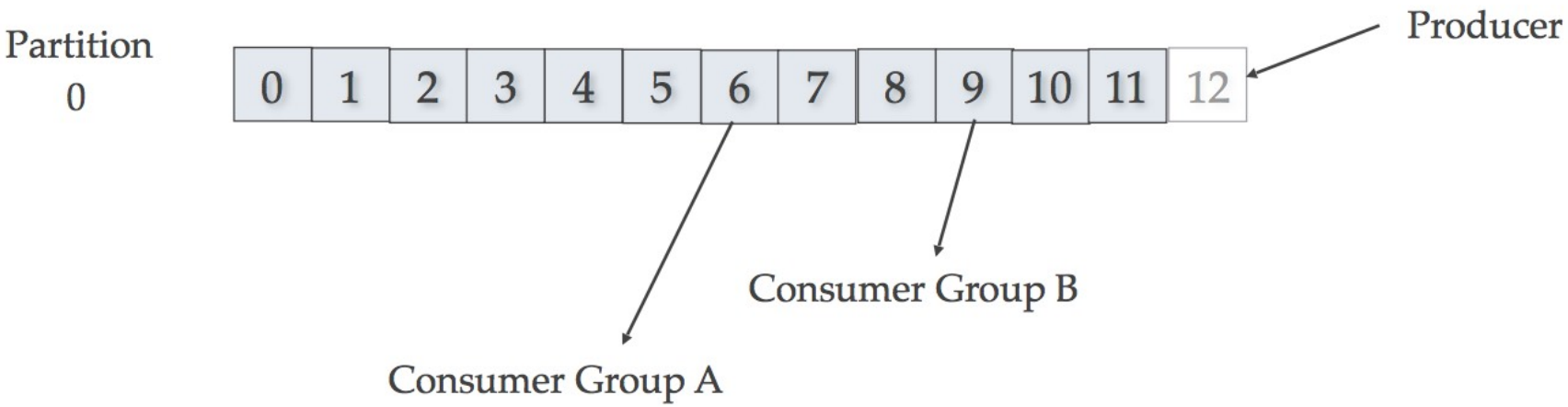
  - Consumer group A, with 2 consumers, reads from a 4-partition topic
  - Consumer group B, with 4 consumers, reads from the same topic

Partition offsets

- **Offset**: messages in the partitions are each assigned a unique (per partition) and sequential id called the *offset*
  - Consumers track their pointers via *(offset, partition, topic)* tuples



Partition offsets



Consumer groups remember offset where they left off.
Consumers groups each have their own offset.

Producer writing to offset 12 of Partition 0 while…
Consumer Group A is reading from offset 6.
Consumer Group B is reading from offset 9.

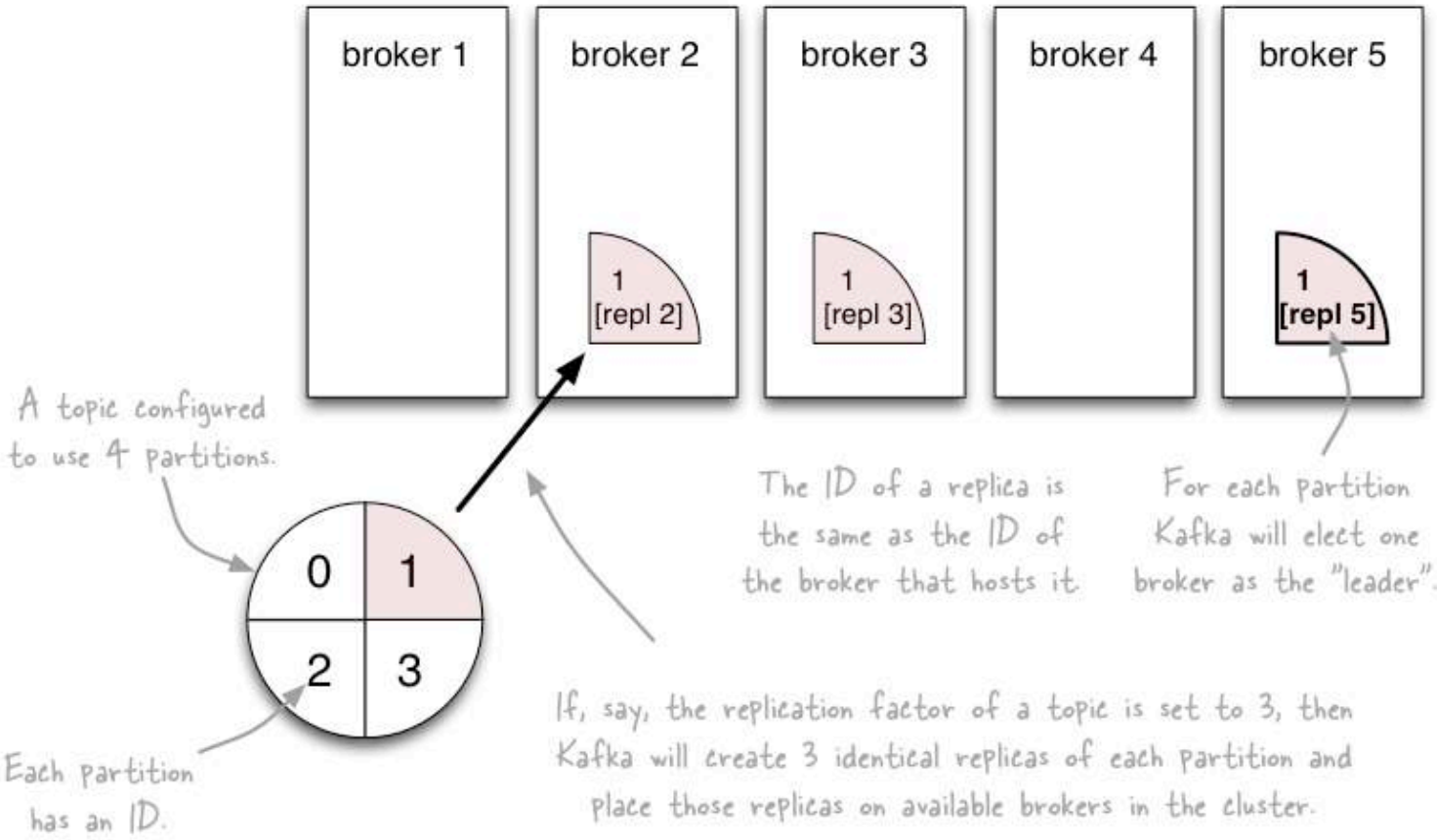
http://www.michael-noll.com/blog/2013/03/13/running-a-multi-broker-apache-kafka-cluster-on-a-single-node/

Replicas of a partition

A partition might be assigned to multiple brokers, which will result in the partition being replicated. This provides redundancy of messages in the partition, such that another broker can take over leadership if there is a broker failure.

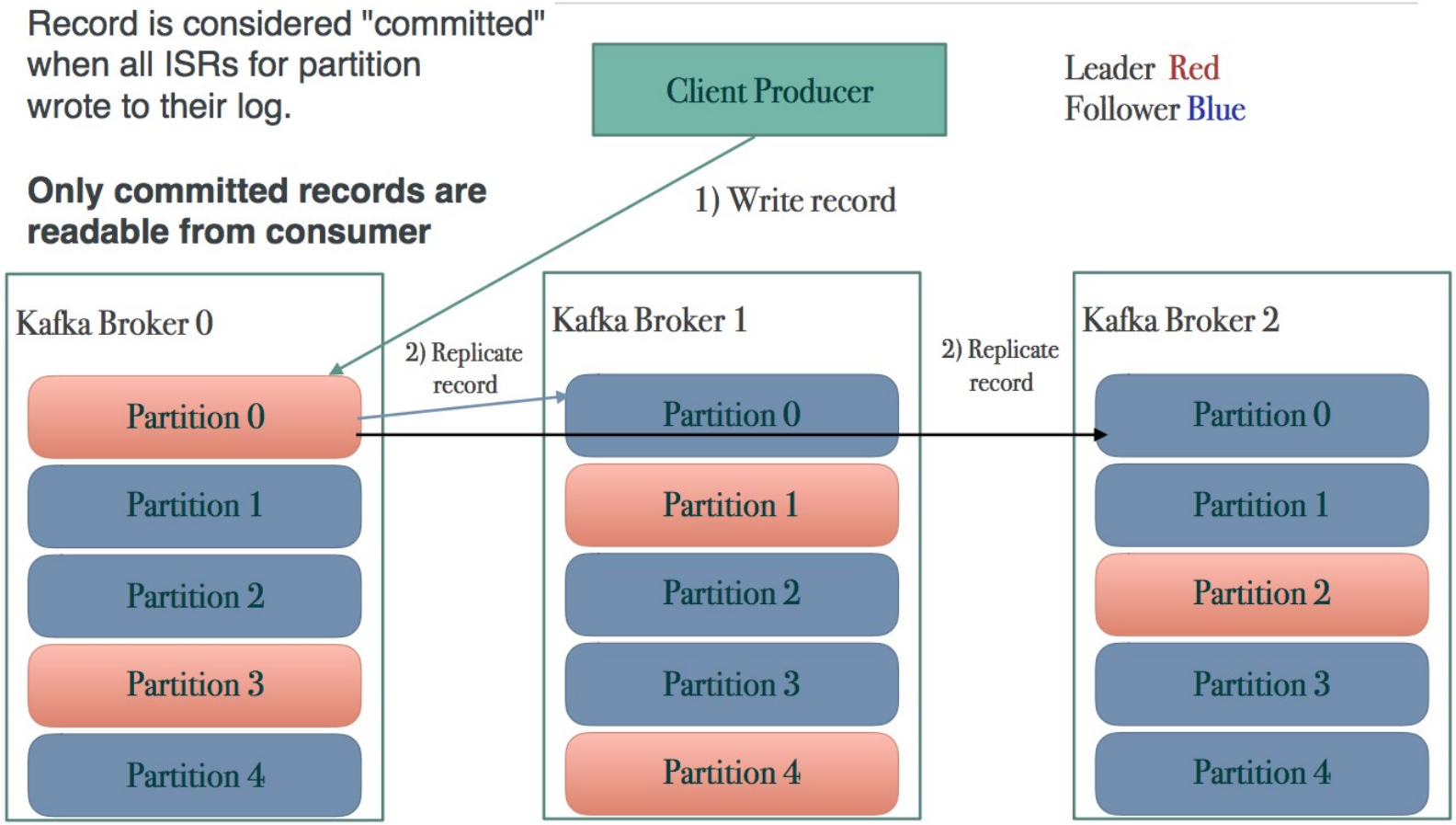
- **Replicas:** "backups" of a partition
  - They exist solely to prevent data loss.
  - Replicas are never read from, never written to.
    - They do NOT help to increase producer or consumer parallelism!
  - Kafka tolerates *(numReplicas - 1)* dead brokers before losing data
    - LinkedIn: numReplicas == 2 ✎ 1 broker can die

Topics vs. Partitions vs. Replicas



http://www.michael-noll.com/blog/2013/03/13/running-a-multi-broker-apache-kafka-cluster-on-a-single-node/

Propagating writes across replicas



ISR = in-sync replica

Inspecting the current state of a topic

- `--describe` the topic

```
$ kafka-topics.sh --zookeeper zookeeper1:2181 --describe --topic zerg.hydra
Topic:zerg2.hydra PartitionCount:3 ReplicationFactor:2 Configs:
  Topic: zerg2.hydra Partition: 0 Leader: 1 Replicas: 1,0 Isr: 1,0
  Topic: zerg2.hydra Partition: 1 Leader: 0 Replicas: 0,1 Isr: 0,1
  Topic: zerg2.hydra Partition: 2 Leader: 1 Replicas: 1,0 Isr: 1,0
```

  - Leader: brokerID of the currently elected leader broker
    - Replica ID's = broker ID's
  - ISR = "in-sync replica", replicas that are in sync with the leader

- In this example:
  - Broker 0 is leader for partition 1.
  - Broker 1 is leader for partitions 0 and 2.
  - All replicas are in-sync with their respective leader partitions.

Scaling

How can Kafka scale if multiple producers and consumers read/write to the same Kafka Topic Log?

- **Writes fast**: sequential writes to filesystem are ***fast*** (700 MB or more per second)
- Scales writes and reads by ***sharding***
  - Topic logs into ***Partitions*** (parts of a Topic Log)
  - Topic logs can be split into multiple Partitions on ***different machines/ different disks***.
  - Multiple Producers can write to different Partitions of the same Topic.
  - Multiple Consumer Groups can read from different partitions efficiently.