Realtime Processing

Real-time processing

- Up until now we've been using batch processing:
    - You submit a MapReduce / Spark / Hive job using Yarn, it executes, and comes back with an answer on screen or writes the output to HDFS
    - You typically run batch jobs in intervals, for example every hour or every day
    - It's the easiest way to get started with Big Data and often people start first with batch and then when there's a need to process the data in smaller time intervals, they add stream processing capabilities

Real-time processing

- Stream processing requires new technologies
  - When ingesting data, the data itself must be temporary stored somewhere
  - HDFS can not quickly read 1 event (e.g. 1 line or 1 record), it's only fast when reading full blocks of data
  - Kafka is a better solution to keep a buffer of our data while it is queued for processing
  - Kafka is a publishing-subscribe (pub-sub) messaging queue rethought as a distributed commit log

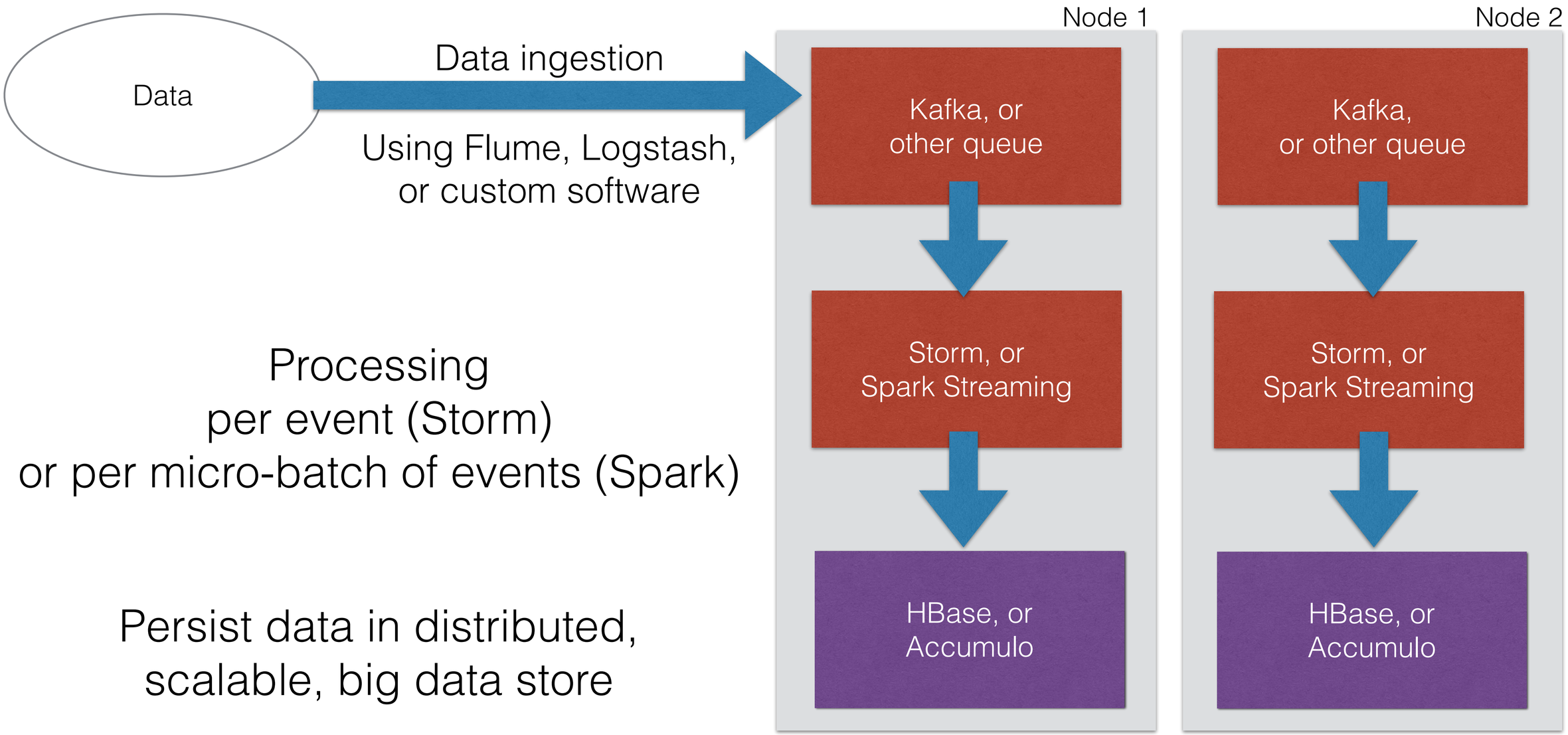
Real-time processing

- Constantly running services within the Hadoop cluster can retrieve the data from a queue system like Kafka
- Every time new data comes in, it needs to be processed in a short timeframe:
  - Storm can process on an event-by-event basis
  - Storm + Trident allows you to do event batching and process multiple events at the same time
  - Spark Streaming currently only support micro-batching, for instance all events within a couple of seconds

Real-time processing

- Once the data has been processed it needs to be stored (persisted)
    - Again HDFS is not a good fit here, the file system hasn't been designed to store a lot of small files. A solution could be to append all events to one or a few files, but then reading one event would be difficult
    - The solution is to use a distributed data store like HBase or Accumulo
    - Those datastores enable fast writes to the datastore and fast reads when you know the row key

Realtime data Ingestion



Processing
per event (Storm)
or per micro-batch of events (Spark)

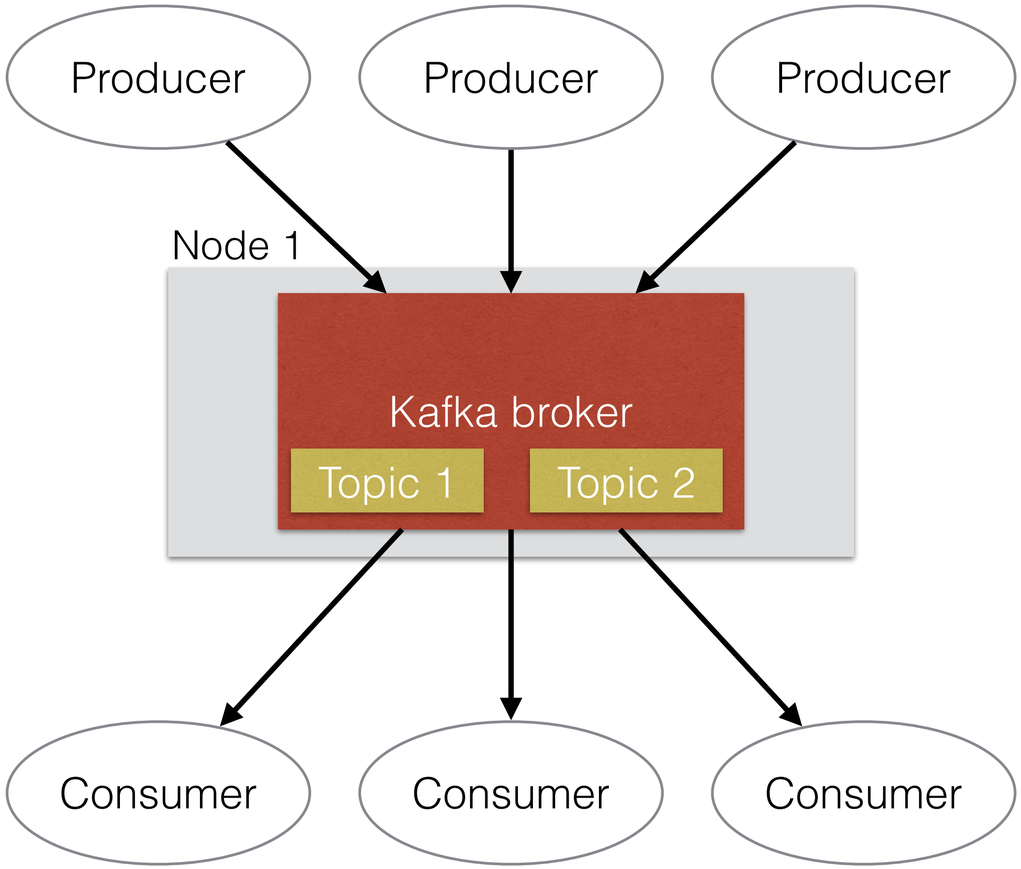
Persist data in distributed,
scalable, big data store

Kafka

Kafka

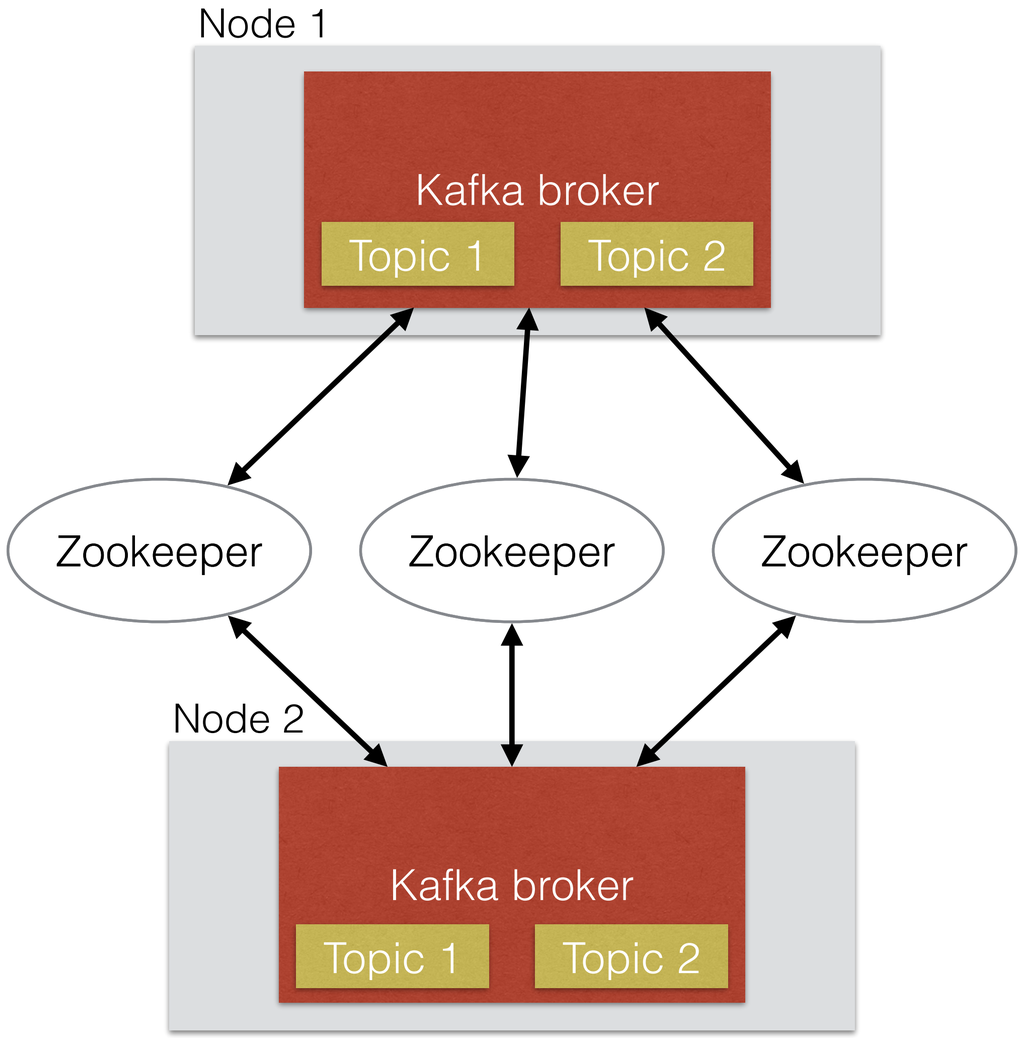
- Kafka is a high throughput distributed messaging system
- It's fast: a single Kafka instance can handle hundreds of megabytes of reads and writes per second from thousands of clients
- Kafka is scalable: it allows to partition data streams to spread your data over multiple machines to exceed the capabilities of a single machine
- Kafka can be transparently expanded without downtime
- It's durable: messages are persisted to disk and replicated within the cluster to avoid data loss. Each Kafka instance can handle TBs of data without performance impact

Kafka Terminology



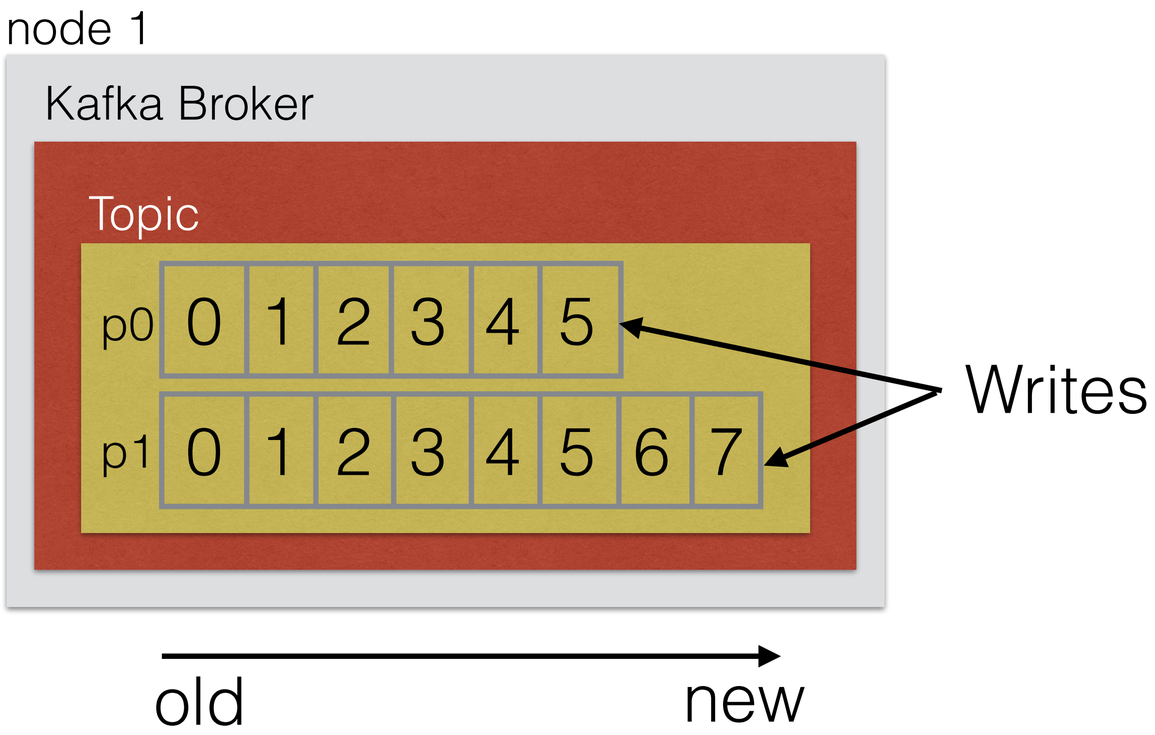
- Kafka maintains feeds of messages in categories called topics.
- The process that sends data (publishes) to Kafka is called the Producer
- The process that reads data (subscribes to topics) is called the consumer
- The Kafka process that runs on our nodes is called the Kafka Broker
- One or multiple brokers together are called the Kafka Cluster

Kafka Terminology



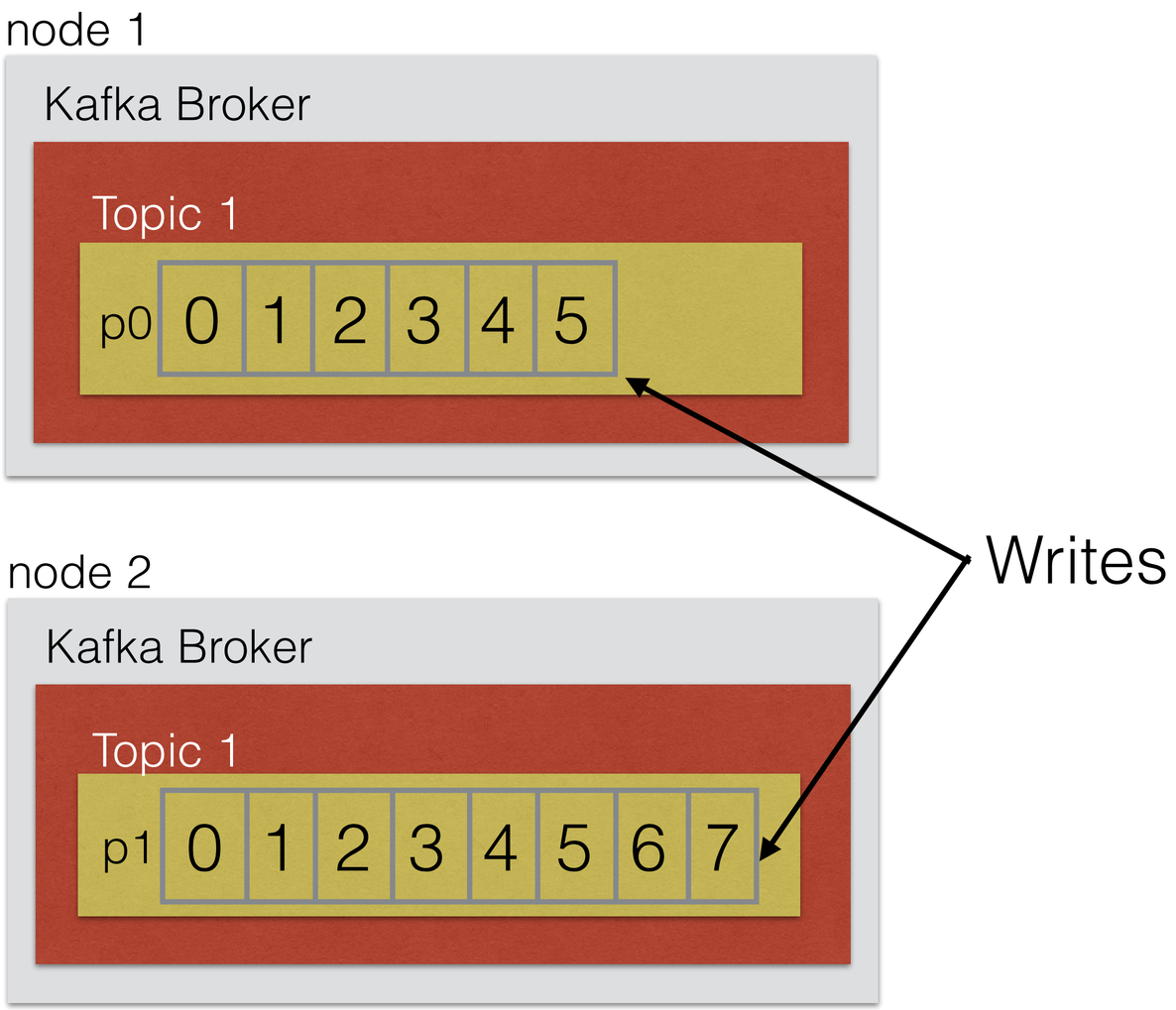
- Kafka uses Zookeeper to know which Kafka brokers are alive and in sync (using Zookeeper heartbeat functionality)
- Zookeeper is also used to keep track of which messages a consumer has read already and which ones not. This is called the offset
- Zookeeper is service that runs on the cluster, where configuration can be stored in a distributed way and that can be used to keep track of which nodes are still alive
- A typical Hadoop cluster has 3 or 5 Zookeeper nodes, providing protection against node failure

Kafka Topics



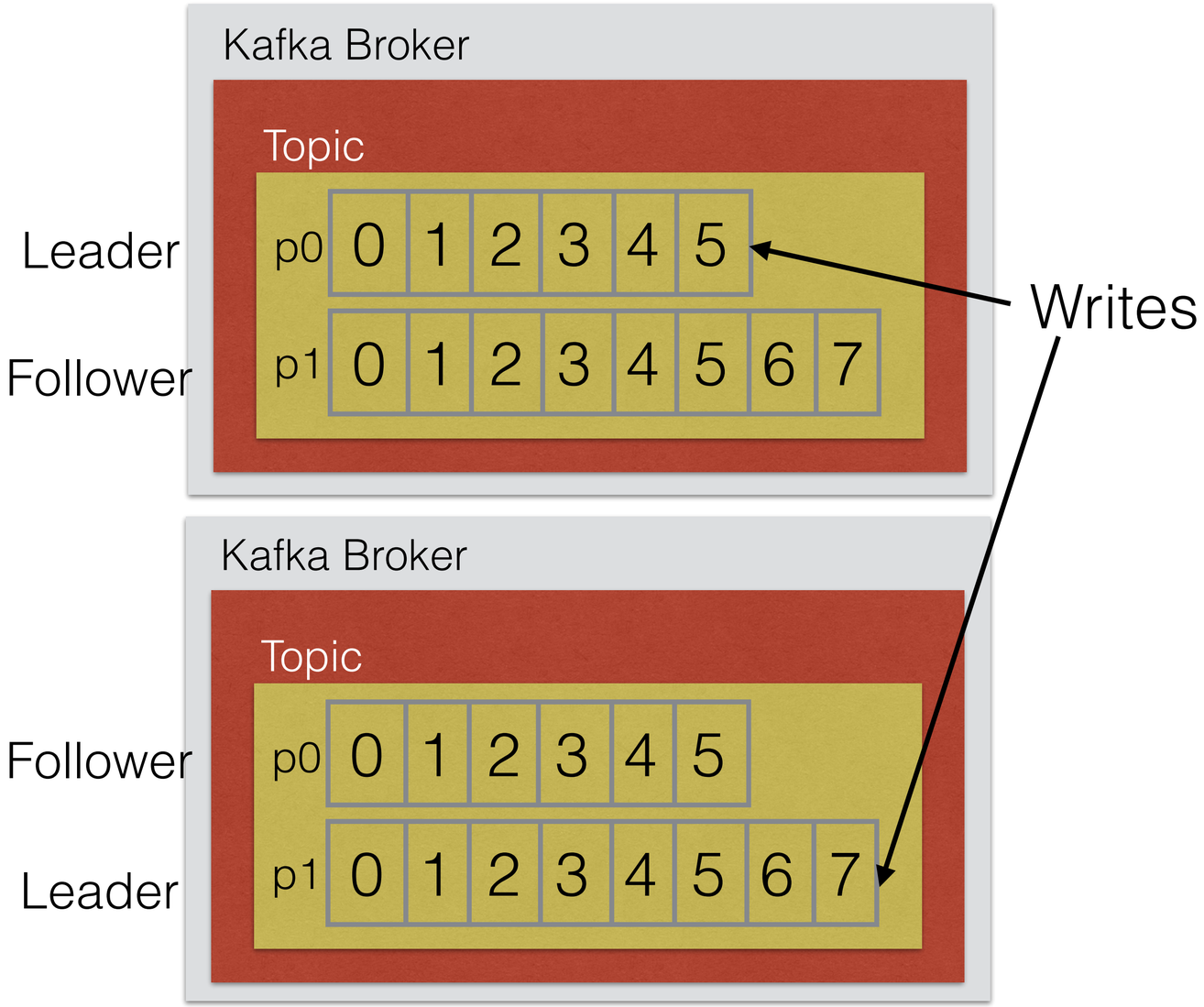
- For each topic, Kafka maintains a partitioned log (p0 and p1 in the picture)
- You can choose how many partitions you want
- Each partition is an ordered, immutable (unchangeable) sequence of messages
- Kafka retains all published messages for a configurable amount of time
- Kafka maintains per consumer the position in the log, called the offset

Kafka Topics



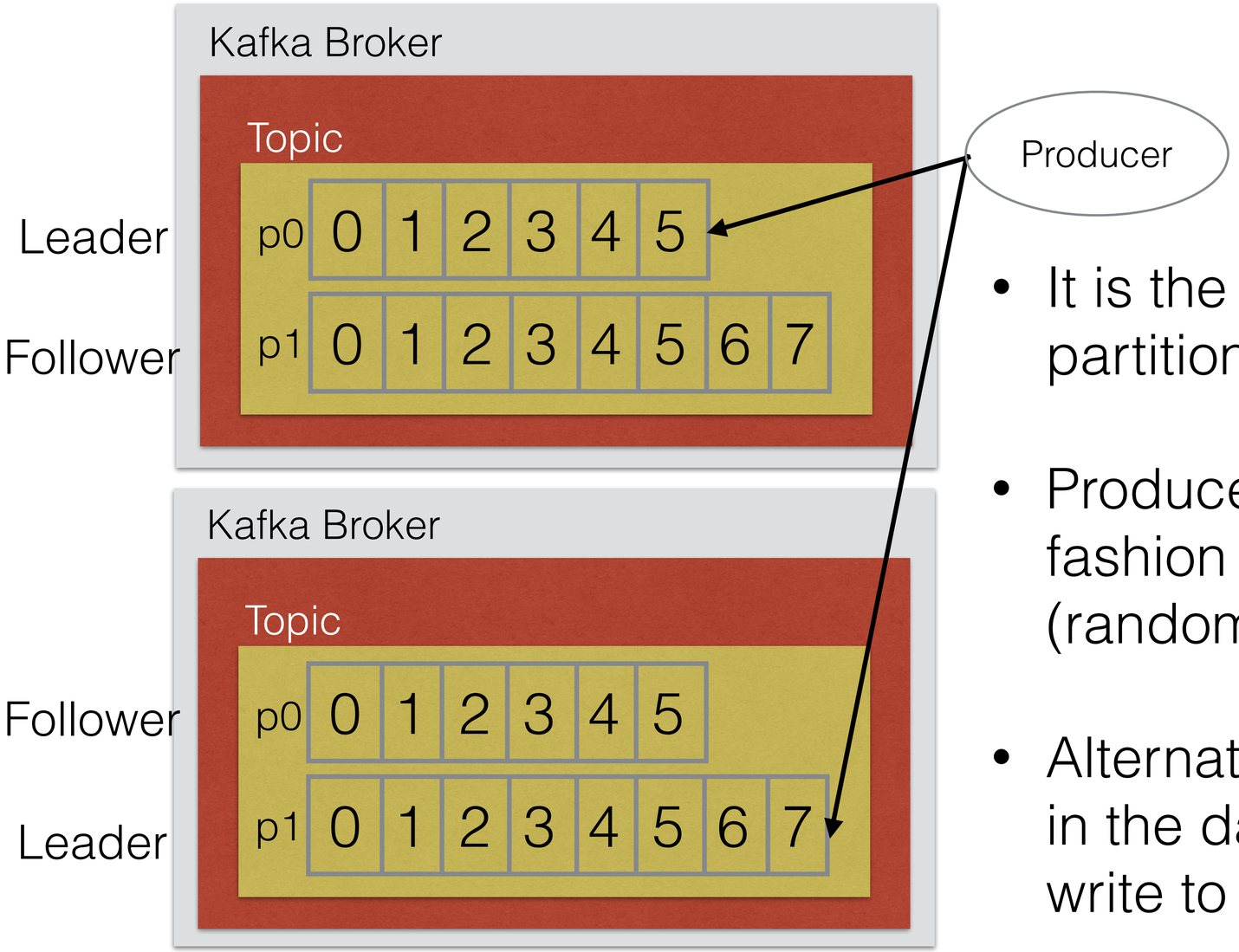
- Partitions allow you to scale beyond the size that will fit on one server
- One topic can have partitions on multiple Kafka Brokers
- Each individual partition must fit on the server that hosts it. You typically have more partitions than nodes, spreading partitions across all your nodes.
- Partitions allow you to execute writes in parallel. The 2 nodes with 1 partition on this slide can handle more read & writes than 1 server with 2 partitions on the previous slide.

Kafka Topics



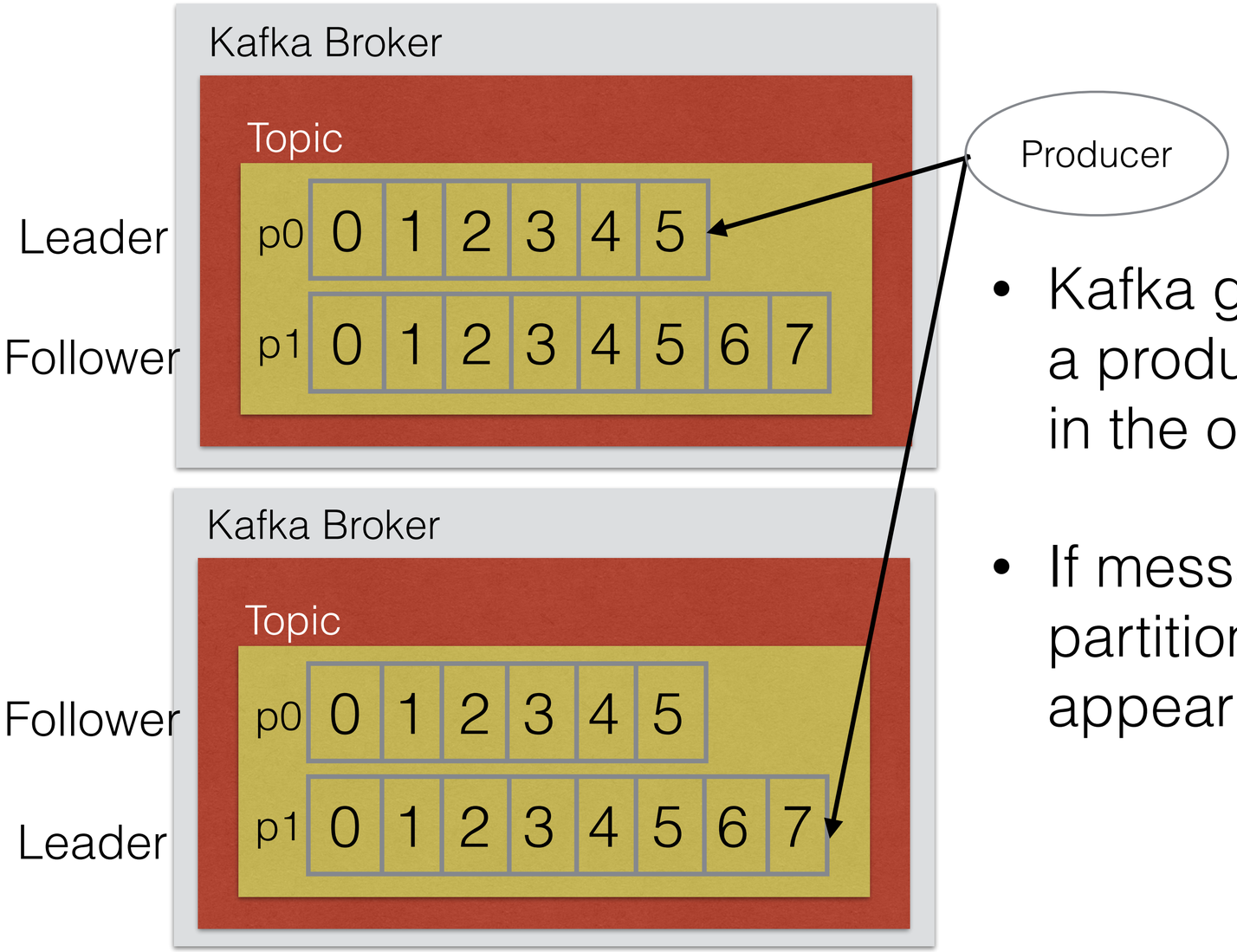
- Replication can be activated on a per topic basis
- Partitions will replicate over the different Kafka brokers
- Writes will always go to the primary partition, called the leader
- The follower partition will retrieve the messages from the leader to be kept “in sync”
- For a topic with replication factor 2, Kafka will tolerate 1 failure without losing any messages (N-1 strategy)

Kafka Topics



- It is the producer process that writes to the partitions
- Producers can write data in a round robin fashion to partition 0 and partition 1 (randomly)
- Alternatively producers can use semantics in the data to make a decision whether to write to p0 or p1

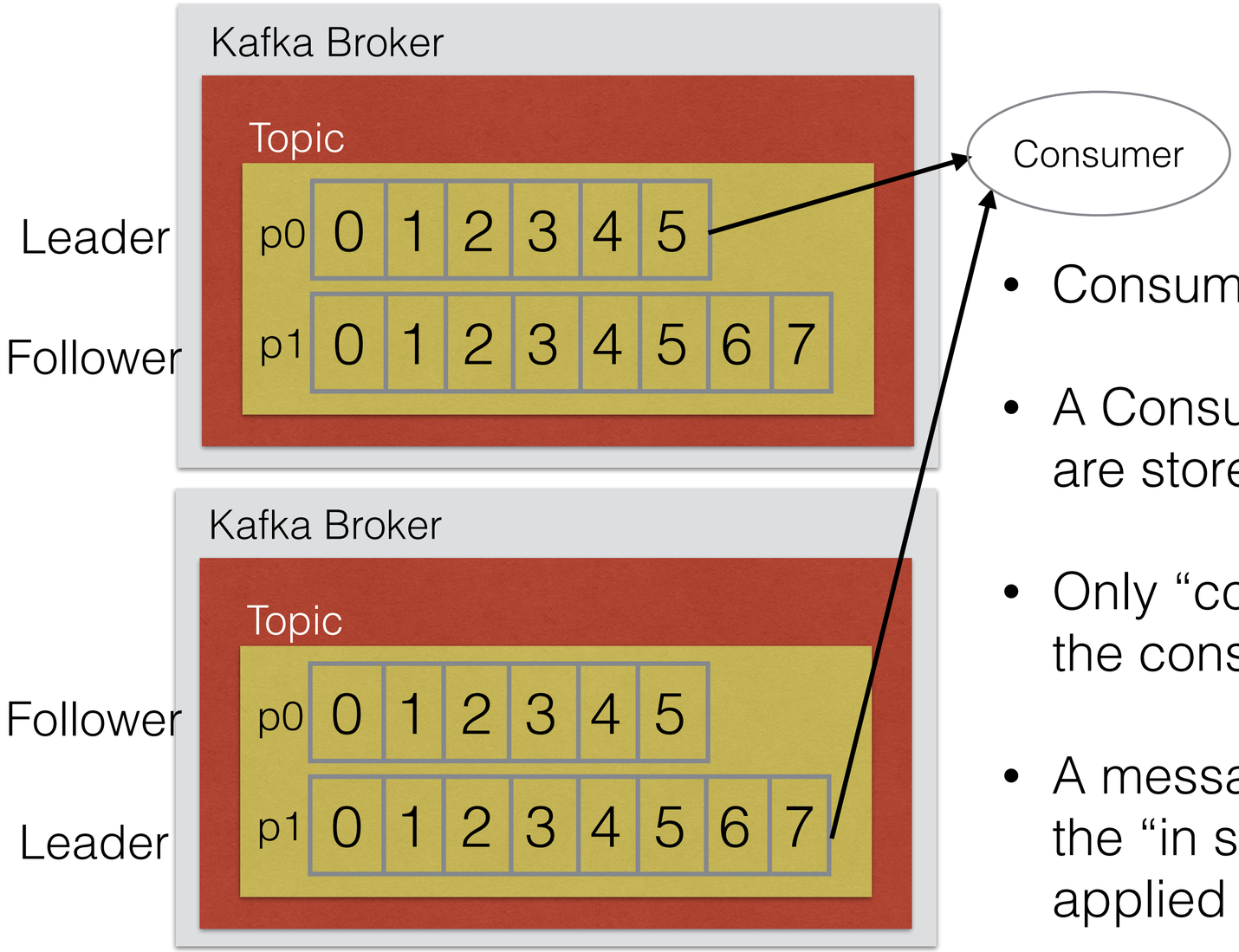
Kafka Topics



- Kafka guarantees that messages sent by a producer to a partition will be appended in the order they are sent

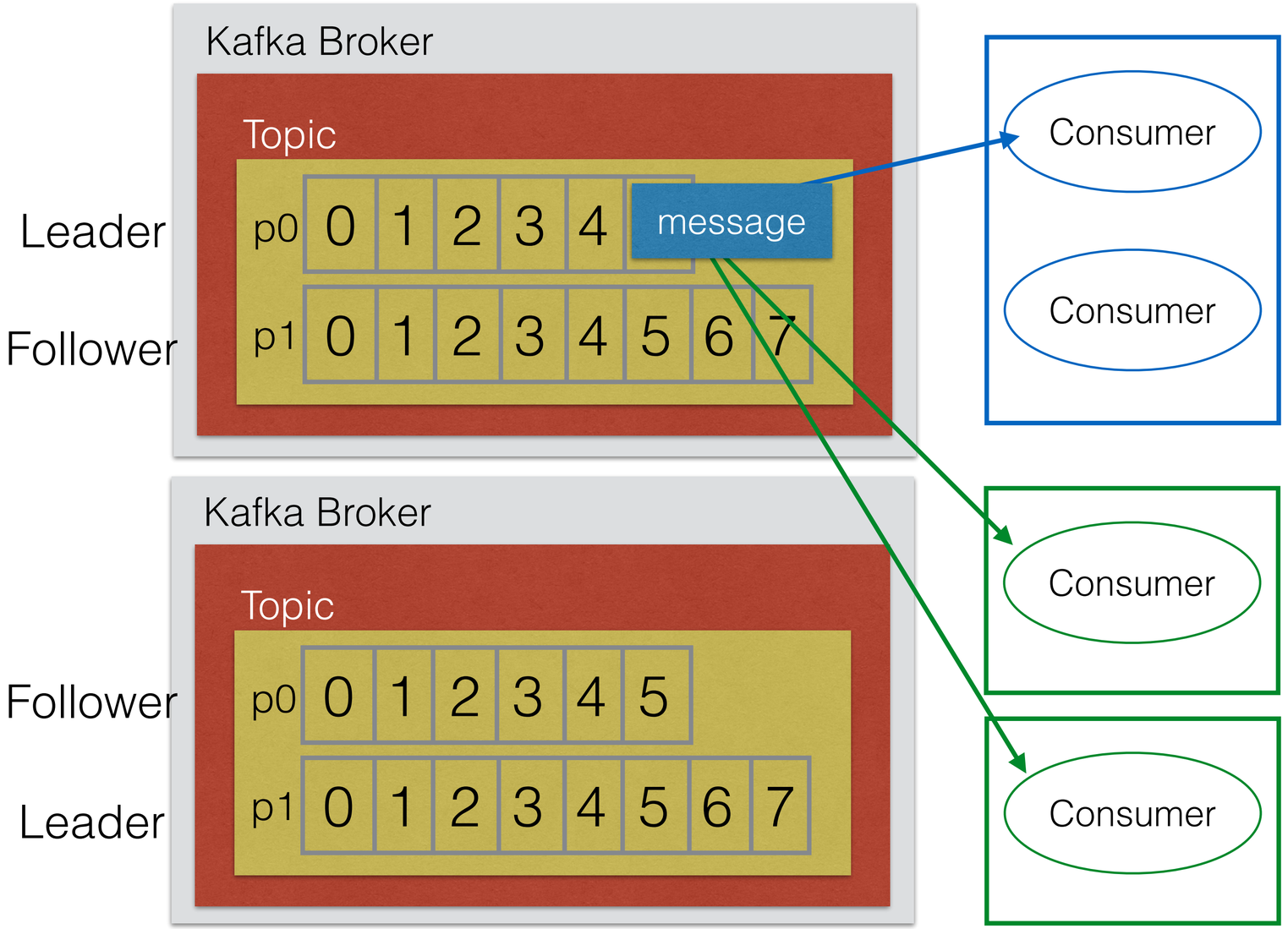
- If message 0 is sent before message 1 to partition 0, then message 1 will always appear after message 0 in the log

Kafka Topics



- Consumers read from the leader partitions
- A Consumer sees messages in the order they are stored in the log
- Only “committed” messages are given out to the consumer
- A message is considered committed when all the “in sync” replicas have the message applied to their logs

Kafka Topics



- Consumer groups can be created
- Each message in a topic is sent to only one consumer in a group
- If you want a queue behavior, put all the consumers in the same group
- If you want a publish-subscribe model, put consumers in multiple groups

Kafka Messages

- Kafka guarantees at-least-once message delivery by default
    - The user can implement an at-most-once message delivery by disabling retries on the producer side
    - For at-most-once message delivery, the consumer also needs to commit its offset before it will process messages, and this on the destination storage system
- Most users keep the at-least-once message delivery by default, and make sure that the processing of messages can be repeatable
    - This is easy to do in distributed databases like HBase, because a data insert will overwrite any existing data (later more about this)

Kafka Messages

- Kafka messages can be compressed
    - Often done when network is a bottleneck
- Rather than compressing every message separately, Kafka can compress messages in batch to achieve better compression
- The batch of messages will be written in compressed form and will only be decompressed by the consumer
- Kafka currently supports gzip and snappy compression algorithms