A Distributed System Case Study: Apache Kafka

High throughput messaging for diverse consumers

As always, this is not a tutorial

Some of the concepts may no longer be part of the current system or implemented as described.

LinkedIn's Requirements

- LinkedIn needs to push billions of messages per day to a wide variety of consumers
- Real-time processing applied to activity streams
  - Keep people engaged with the site through social network-supplied content.
- Asynchronous processing
  - Find people I may want to connect to
  - Find topics that may be interesting to me
  - Identify advertisements that I may be interested in
- System logs: Identify problems with operations

Machine learning is critical to all of these. But is OK to lose some data.

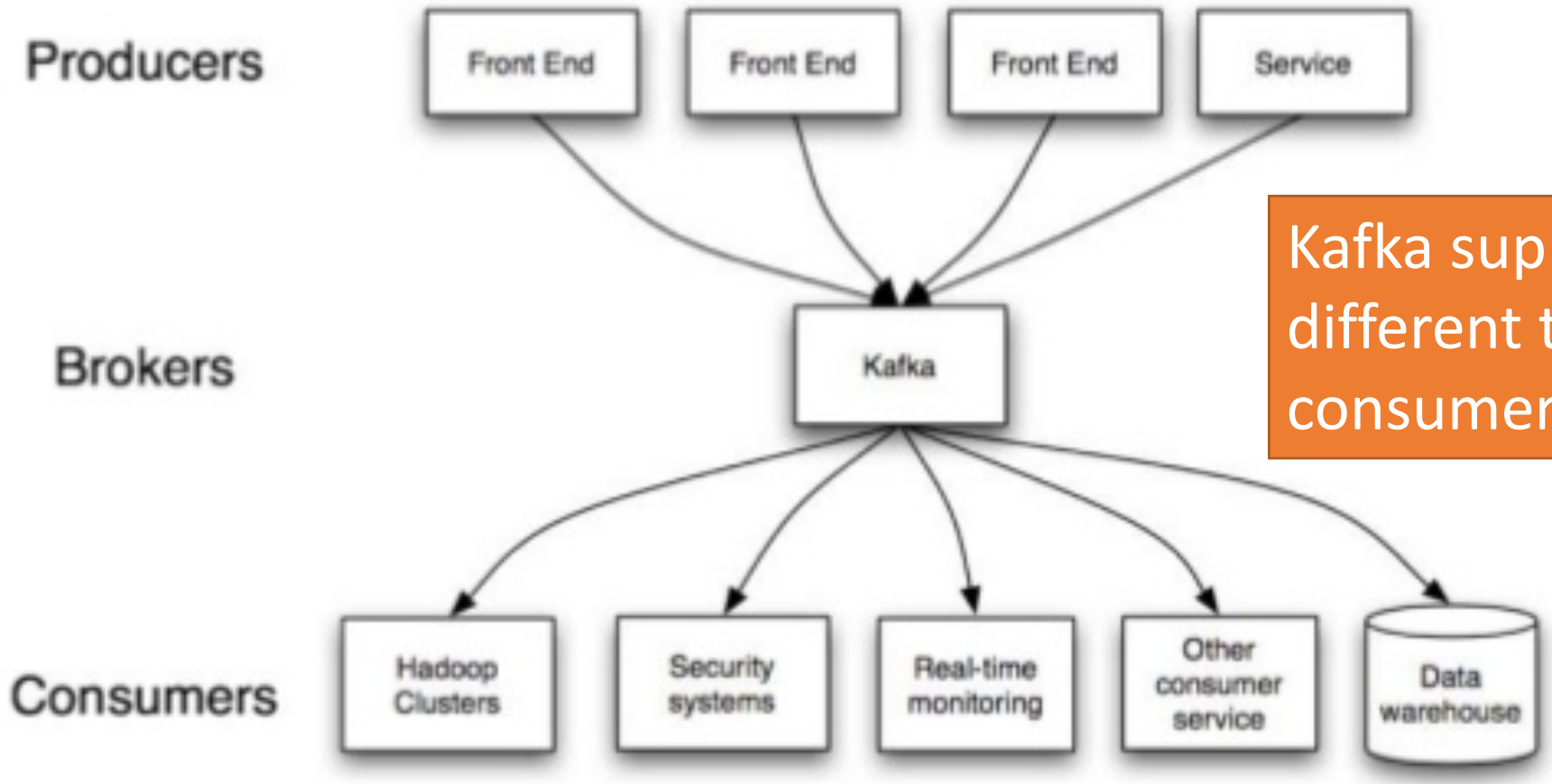
Consequences of Use Case Requirements

- Consumers must decide when to pull the data
  - Fast or slow, small or large
- This means that the messaging system needs to store a lot of data (TBs)
  - This is not what traditional message systems are designed to do.
  - Kafka will need an efficient way to find the requested message
- Virtue from Necessity: support message rewind and replay
  - This is not a normal operation for a queue, which removes messages after they are delivered.
  - Treat the accumulated, ordered messages as input for a state machine

Apache Kafka and Related Work

- Apache Kafka is a hybrid of a log aggregator and a messaging system
- Messaging systems
  - Usually assume low latency of delivery. Messages are delivered quickly
    - This doesn't work for batch systems
  - Have complicated delivery guarantees that aren't needed for real-time data
  - Are not designed for high throughput techniques such as message batching
  - Weakly designed as distributed systems: consistency rather than availability (CAP)
- Distributed Log Aggregators (Scribe, Flume, Hedwig)
  - Are only for server logs
  - Only support push model: producer sets the rate
  - Kafka use case: pull model lets consumers set the rate, support rewind

Kafka decouples data-pipelines



Kafka supports many different types of consumers

https://image.slidesharecdn.com/current-141113081750-conversion-gate02/95/current-and-future-of-apache-kafka-9-638.jpg?cb=1415866728

Apache Kafka Terminology: Topic-Based Publish-Subscribe

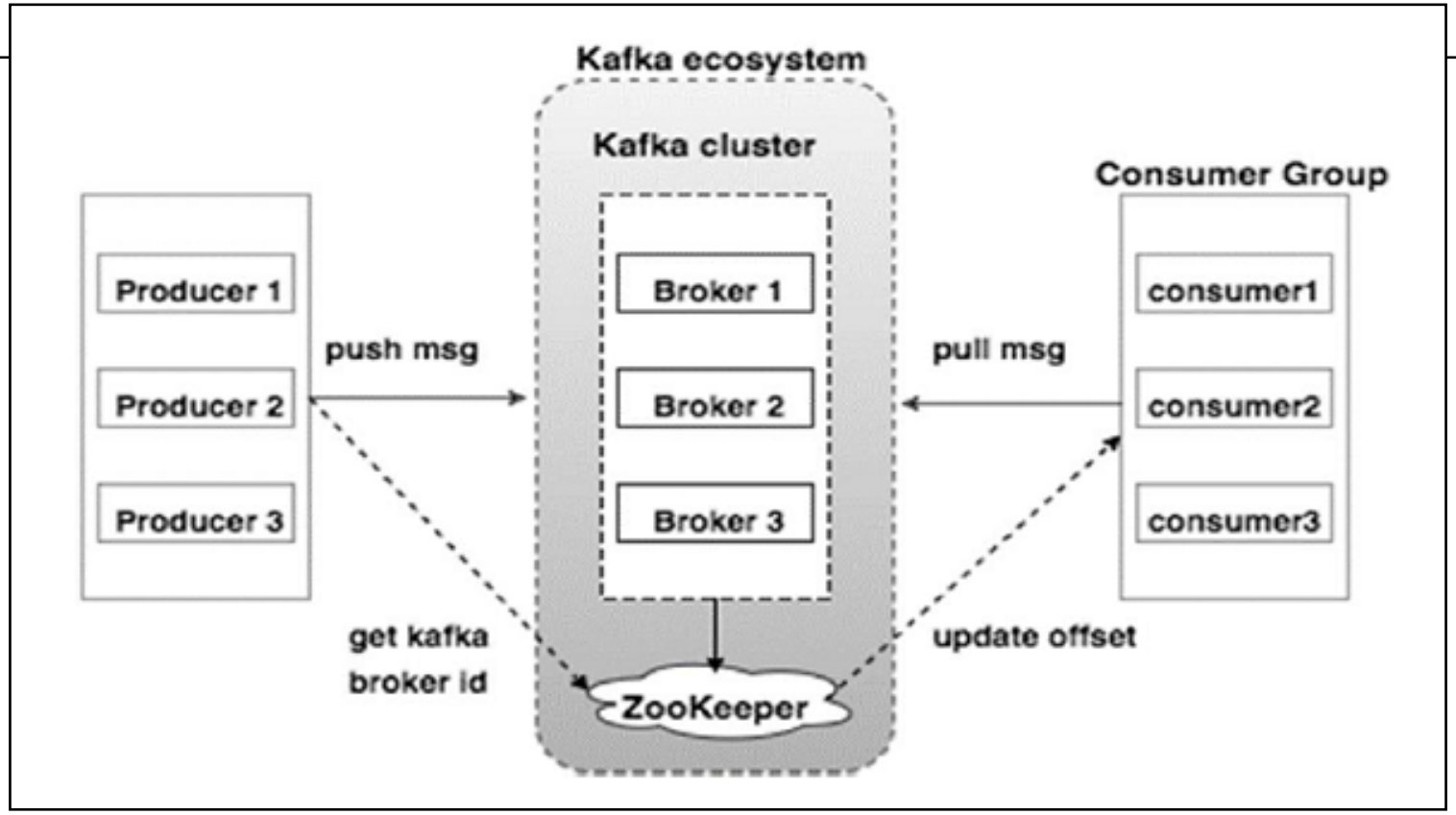
Component	Description
Topic	The label for a stream of messages of a particular type.
Producer	An entity that publishes to a topic by sending messages to a broker
Broker	An entity on a network that receives, stores, and routes messages.
Consumer	An entity that subscribes to one or more topics. Kafka generalizes this to **Consumer Groups**

Kafka Brokers

Connecting message producers to consumers

Kafka Uses Clusters of Brokers

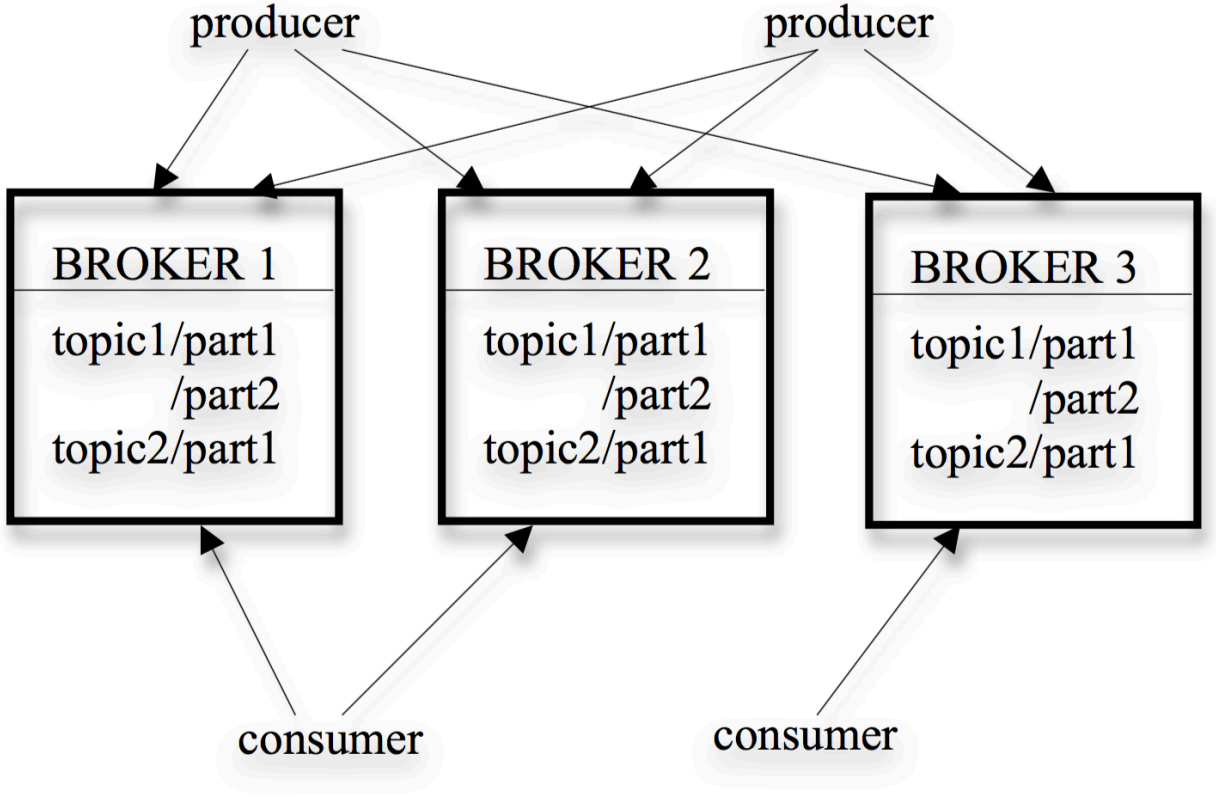
- Kafka is run as a cluster of brokers on one or more servers.



Note use of Zookeeper. Zookeeper also used by Producers and Consumers

Distributed Brokers

- Kafka brokers are designed to be distributed
  - Multiple brokers
  - Messages to each topic are stored in partitions
  - Partitions are allocated across brokers
- Partitions are stored as multiple segment files on a specific broker.

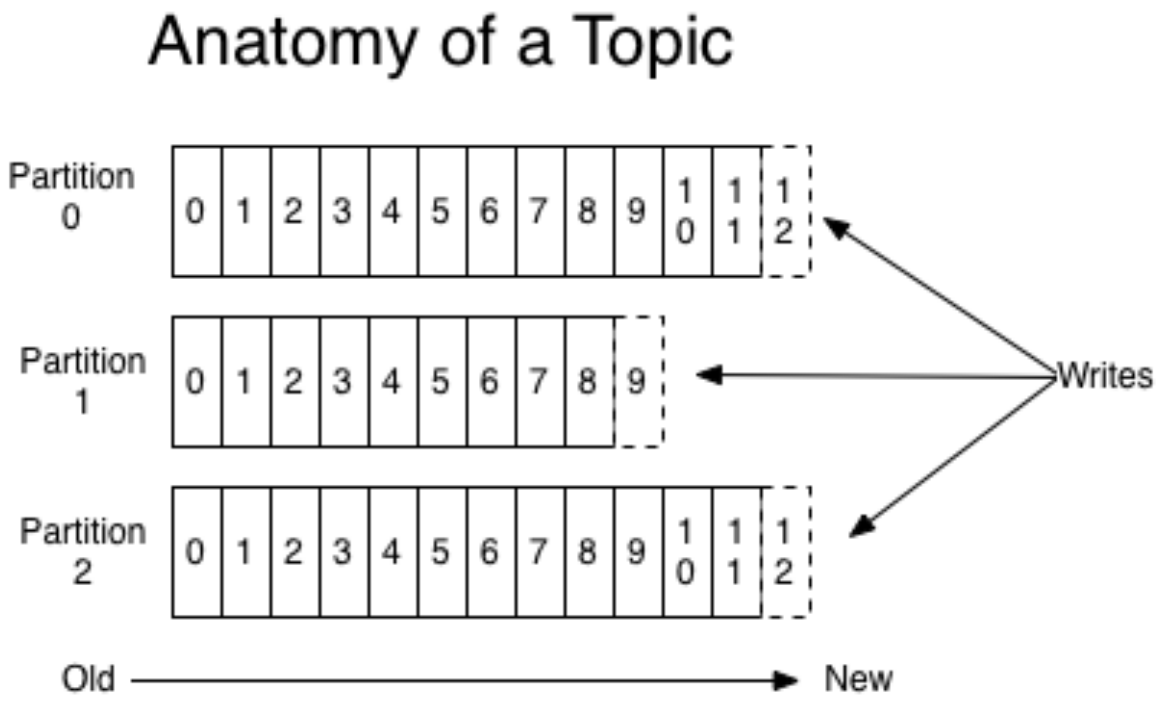


Topics and Partitions

Topics are broken up into **partitions** that span multiple brokers

Kafka Partitions

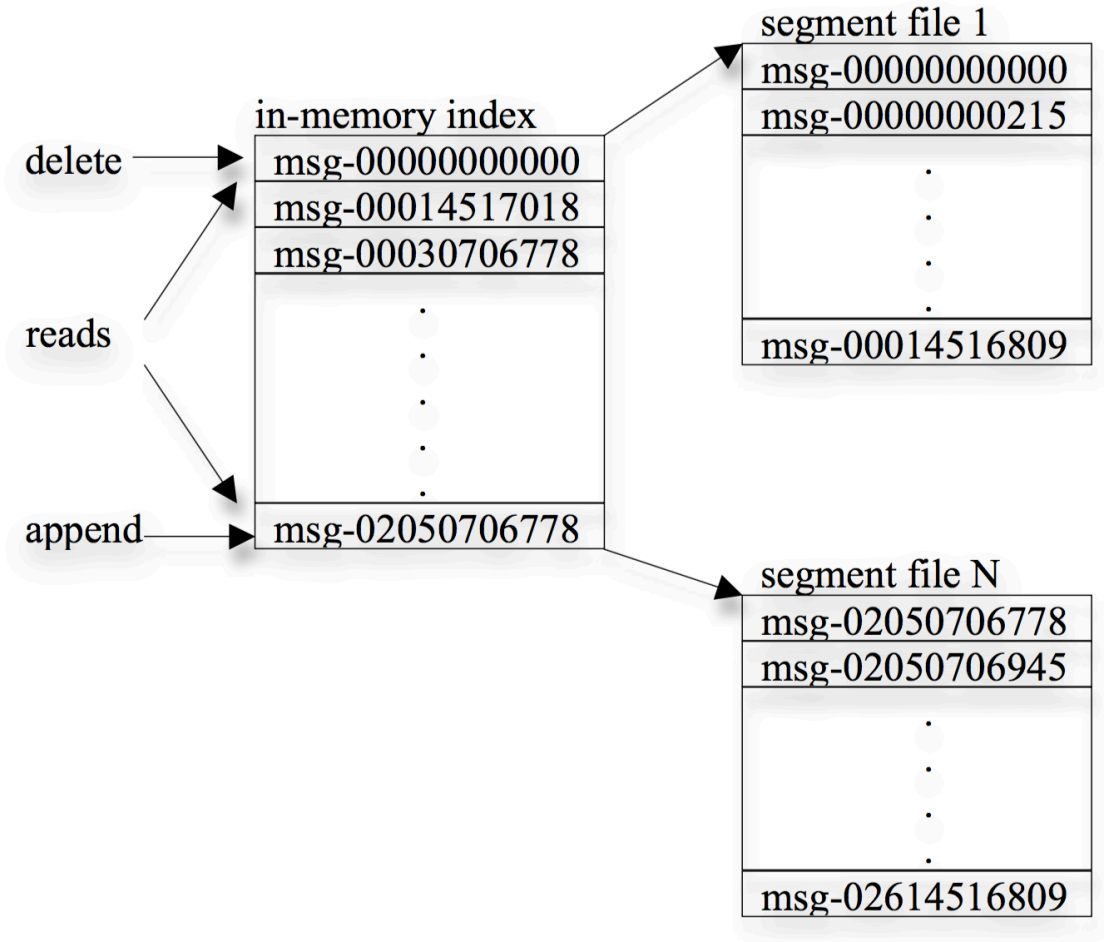
- Kafka topics are divided into *partitions*.
- Partitions parallelize topics by splitting the data across multiple brokers
- Each partition can be placed on a separate machine
  - Allows multiple consumers to read from a topic in parallel.
  - Topic content size can be larger than physical storage on any one broker
- Consumers can also be parallelized to read from multiple partitions



Kafka guarantees all messages in a partition are time-ordered, but it does not guarantee order across partitions. Choose partitioning strategies accordingly.

Sequential, Deterministic Lookups

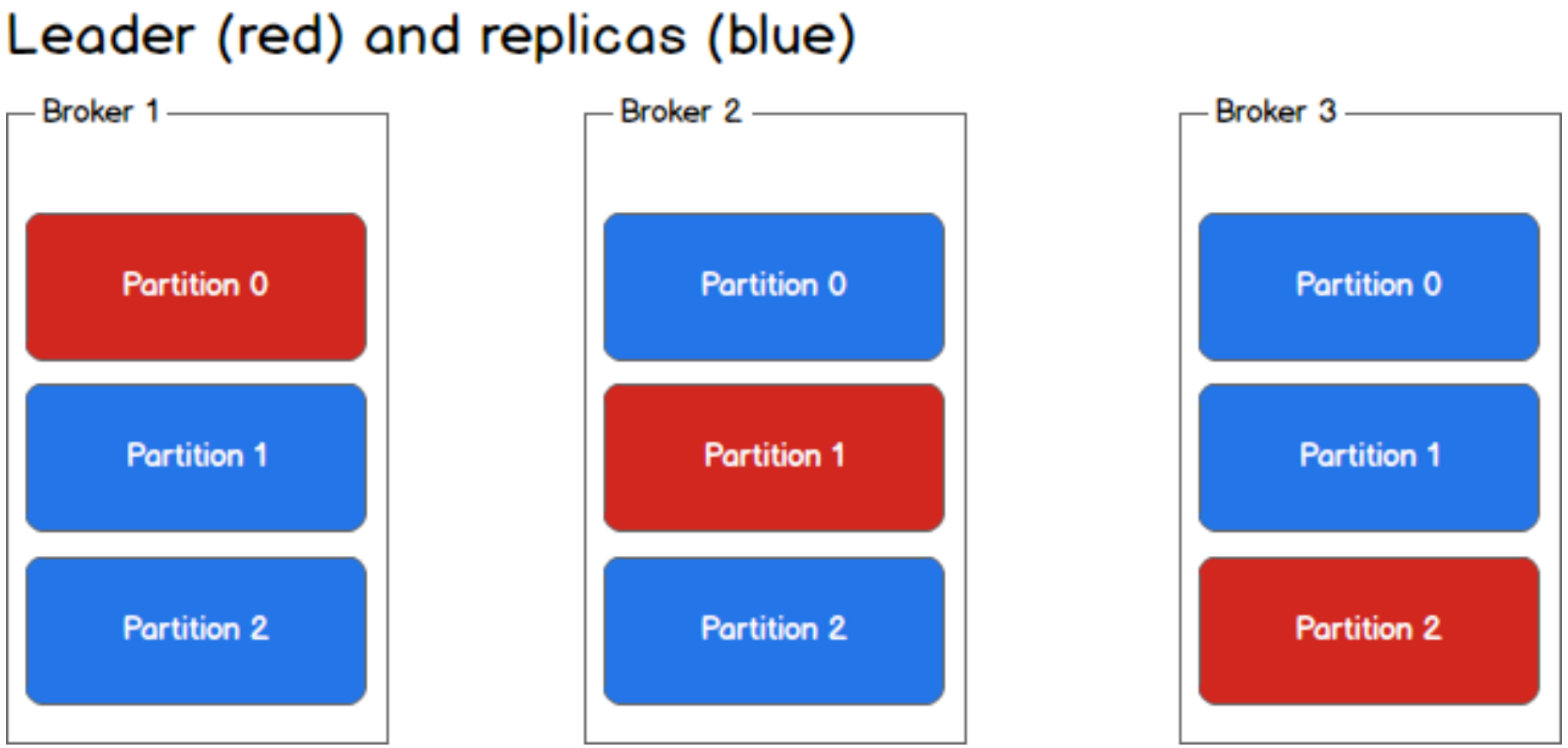
- Random access is anathema to Kafka's goals for high throughput.
- Partitions consist of one or more segment files
- Each message stored in a segment file has a local ID that is determined by the size of all the messages that come before.
- An index stores the message ID of the first message in a segment



Aside: Write-Ahead Logging

- Write-Ahead Logging: this is a technique of writing your file first and then having your broker read the file.
  - This is the reverse of the way logging normally works
  - Why? If the broker needs to be restarted, it reads it log to recover its state.
  - You don’t need to worry so much about lost messages from publishers.
- Kafka uses the file system
  - Linux file systems already have many sophisticated features for balancing in-memory versus on-disk files

Kafka Partitions Are Replicated



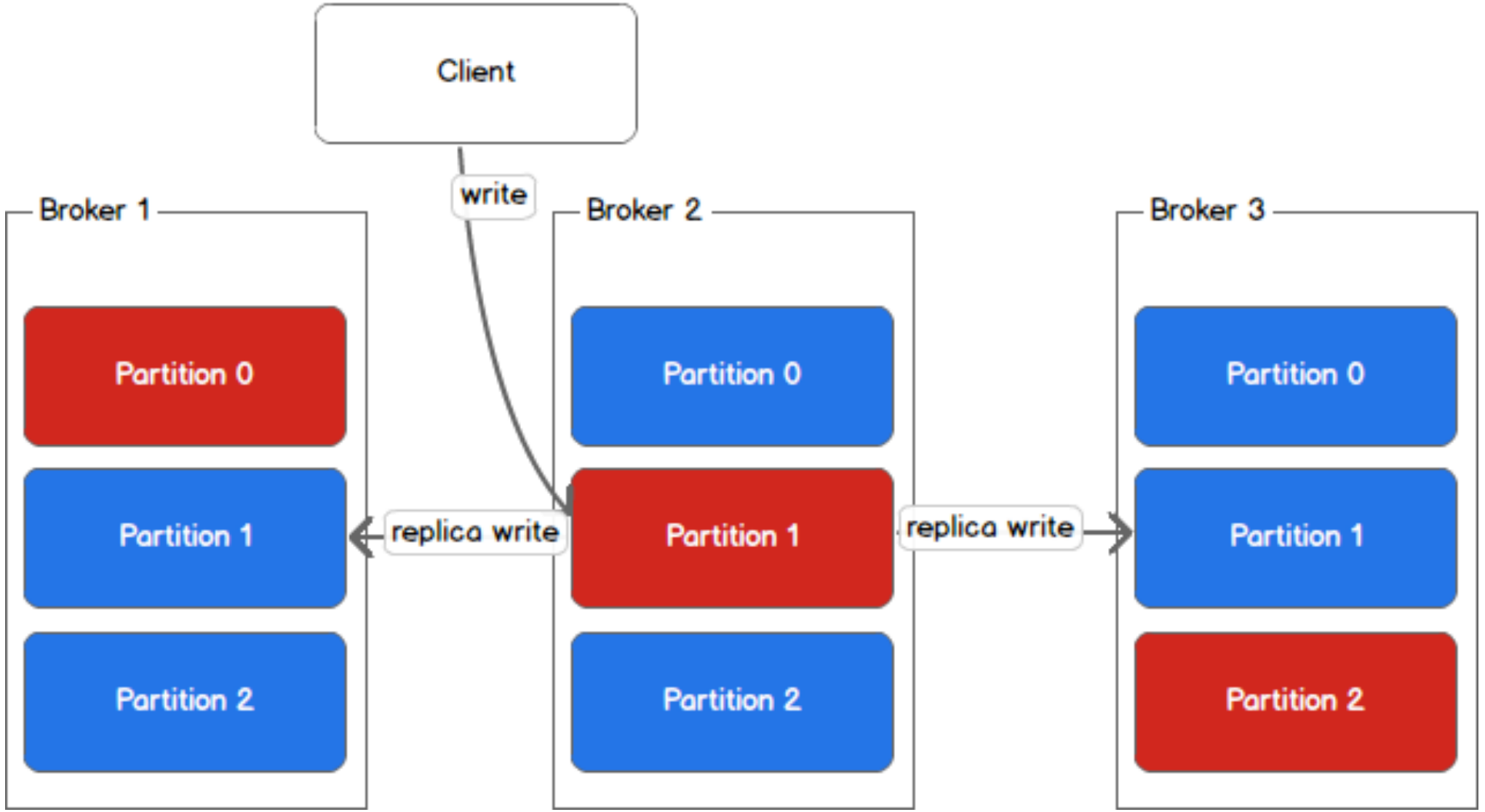
If a leader for a partition replica fails, a follower becomes the new leader

Producers

Kafka Producers

- Producers publish data to the topics of their choice.
- The producer is responsible for choosing which record to assign to which partition within the topic.
- This can be done in a round-robin fashion simply to balance load
  - Other distribution strategies can be used.
- Producers write to the partition’s leader.
  - The broker acting as lead for that partition replicates it to other brokers

Leader (red) and replicas (blue)



A producer writes to Partition 1 of a topic. Broker 2 is the leader. It writes replicas to Brokers 1 and 3

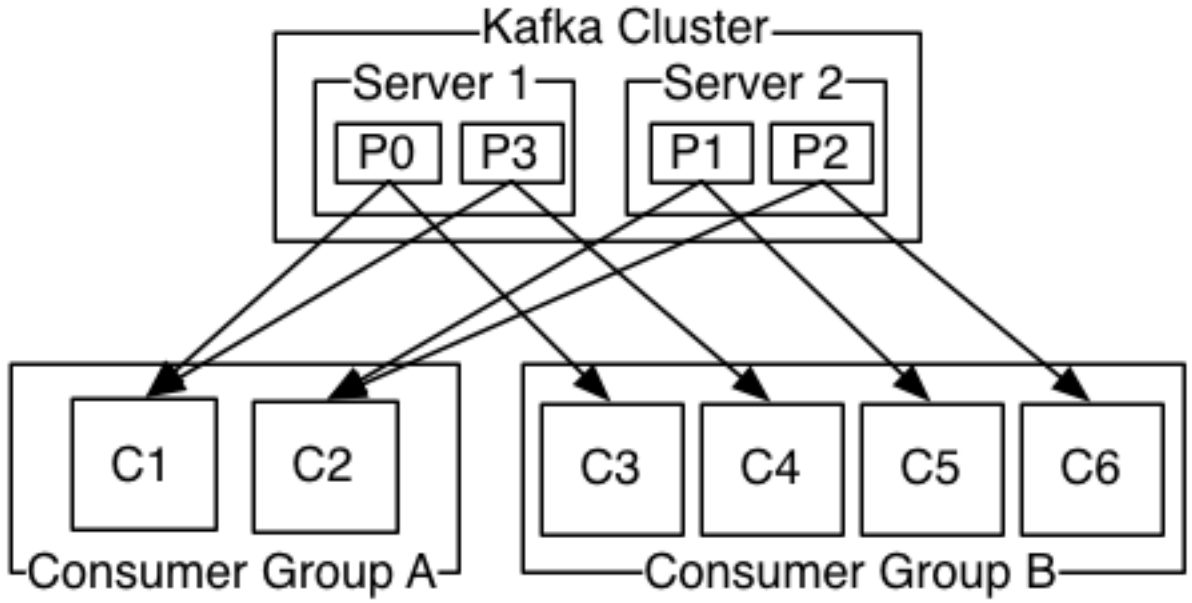
Consumer Groups

Kafka Consumer Groups

- Consumer groups contain one or more consumers of a particular topic.
  - Many consumer groups can subscribe to the same topic.
  - A consumer group is whatever is useful for a particular consuming application
- Only one member of a consumer group consumes the messages in a partition
  - Avoid locking and other state management issues
- Kafka wants to divide messages stored on brokers evenly among consumers for load balancing
  - But keep it simple, avoid complicated coordination.

Consumers and Consumer Groups

- Kafka lets you collect consumers into consumer groups
- In a group, each consumer instance is associated with a specific partition.
- Collectively, a group receives all messages on a topic.
- If the group expands or contracts, Kafka will rebalance.



Consumer groups resemble queues

Consumer Group Scenarios

- Consumer Load Balancing: All consumers are in one group
  - Consumer Group B, previous slide
- Broadcast: each consumer is its own group
  - Each consumer receives messages from all partitions
- Broker Load Balancing: N(Brokers) > N(Consumer Groups)
  - Consumer Group A, previous slide
  - But messages between partitions may not be ordered
- Consumer ordering: N(Partitions) == N(Consumers in a Group)
  - Each member gets messages from only one partition

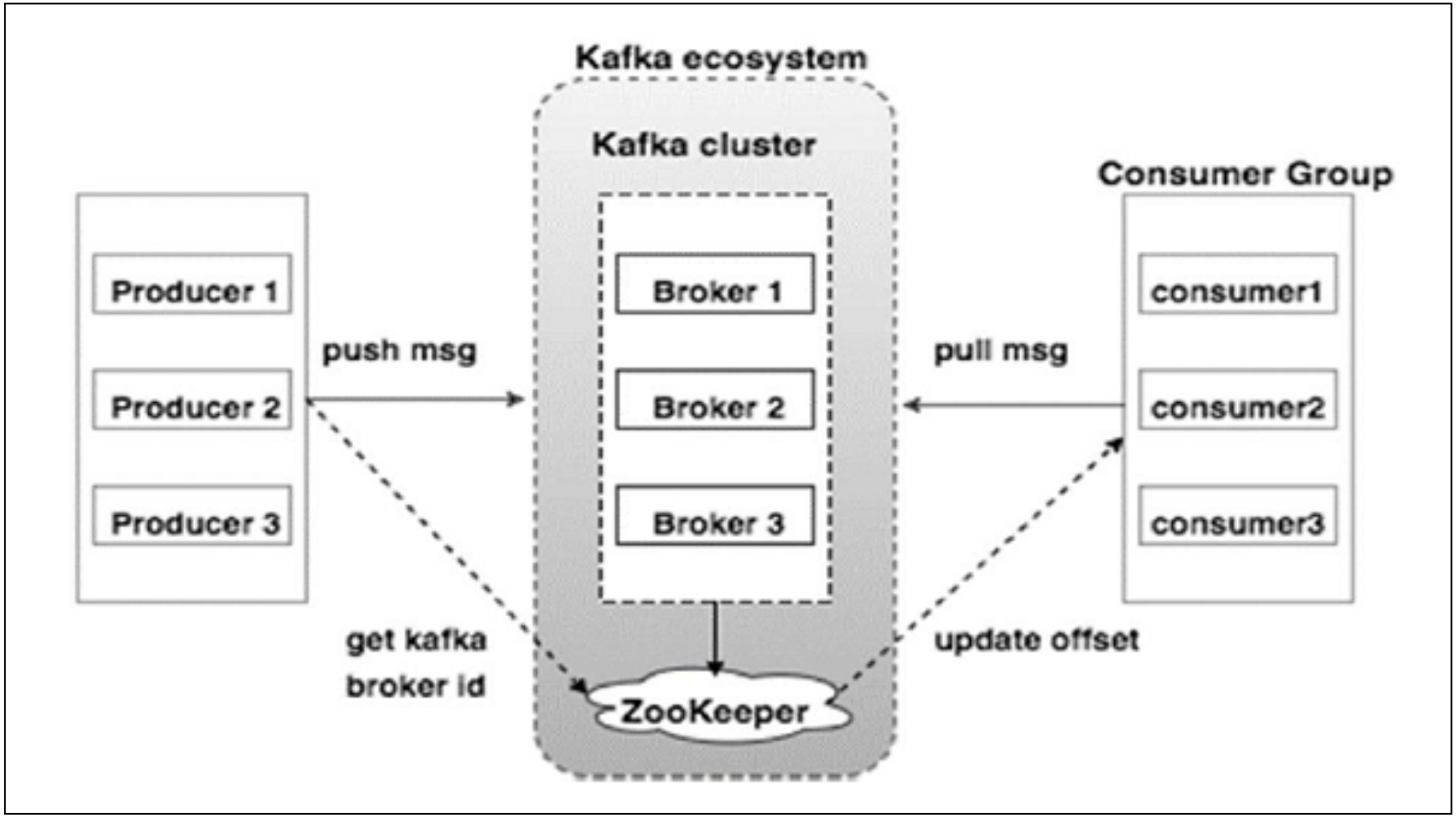
We assume round-robin message distribution to partitions. Note message order is preserved only within a partition

Rewinding and Replaying Messages

- Kafka persistently stores messages much longer than conventional messaging systems
  - Doesn’t assume low-latency delivery.
- The state of a topic is the message order, stored in partition files.
- A consumer can request the same messages many times if it needs to.
  - Why? Rollback. A consumer may have had a bug, so fix the bug and consume the message again with the corrected code.
    - Recall Blue-Green deployments
  - Or the consumer may have crashed before processing the message
  - This is NOT a typical queue pattern.
- Rewinding is much more straightforward in a pull-based architecture.

Kafka and Zookeeper

Managing brokers, consumers, and producers


Kafka ecosystem
Kafka cluster
Consumer Group
Producer 1
Producer 2
Producer 3
push msg
Broker 1
Broker 2
Broker 3
pull msg
consumer1
consumer2
consumer3
get kafka
broker id
ZooKeeper
update offset

Zookeeper Registry	Description	Node Type
Broker Registry	Contains brokers' host names, ports, topics, and partitions. Used by the brokers to coordinate themselves. Ex: deal with a broker failure.	EPHEMERAL
Consumer Registry	Contains the consumer groups and their constituent consumers.	EPHEMERAL
Ownership Registry	Contains the ID of the consumer of a particular consumer group that is reading all the messages. This is the "owner".	EPHEMERAL
Offset Registry	Stores the last consumed message in a partition for a particular consumer group.	PERSISTENT

Each consumer places a watch on the broker registry and the consumer registry and will be notified if anything changes.

Delivery Guarantees

- Kafka chooses “at least once” delivery.
  - It is up to the consuming application to know what to do with duplicates
  - Duplicates are rare, occur when an “owning” consumer crashes and is replaced
  - **Two-phase commits** are the classic way to ensure “exactly once” delivery.
- Messages from a specific partition are guaranteed to come in order.
- Kafka stores a **CRC** (a hash) for each message in the log to check for I/O errors

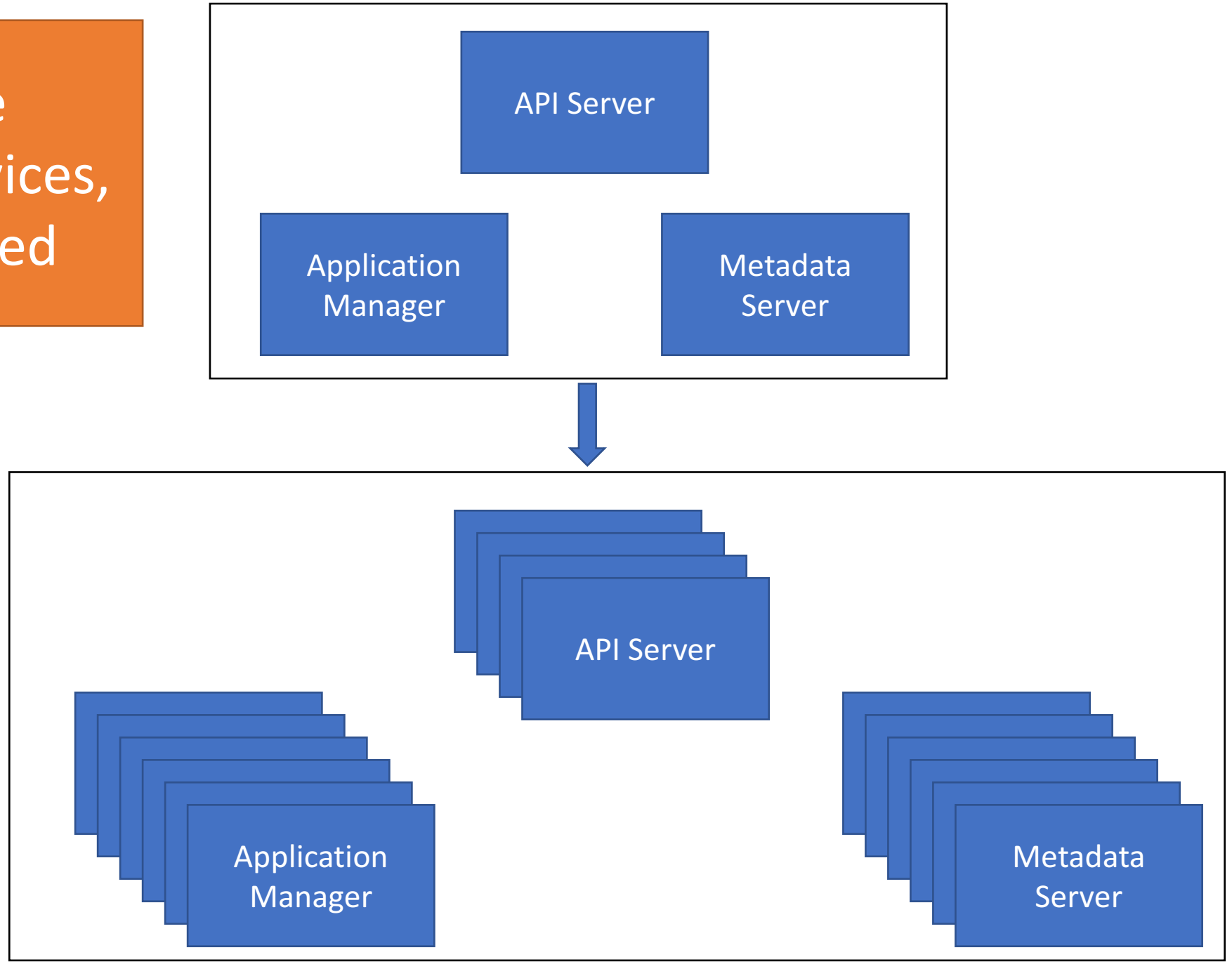
What About the Message Payload?

- Apache Kafka supports clients in multiple programming languages.
- This means that the message must be serialized in a programming language-neutral format.
- You can make your own with JSON or XML
- Kafka also supports Apache Avro, which is a schema-based binary serialization format.
  - Compare Avro with Apache Thrift and Protobuf
- Efficient message formats are essential for high throughput systems

Kafka, Airavata, and Microservices

Some thought exercises

Three microservices, replicated



Messaging and Microservices

- In Assignment 2, we that message queues are a good way to deliver messages to service replicas.
  - Each replica of a service gets a message
- Topics are a good way to associate message types with different service groupings.
  - Application manager and metadata manager
- In Kafka, we could put each of these into a consumer group
- However, work queues are very important to Airavata
  - This is not directly implemented in Kafka, so we would need to implement
  - This is probably not a good strategy.
- We could always just use Kafka as a log aggregator.

Some General Distributed Systems Principals

Kafka, logs, and REST

Log-Centric Architecture

- Distributed servers use a replicated log to maintain a consistent state
- The log records system states as sequential messages.
- New servers can be added to expand the system or replace malfunctioning servers by reading the log
  - No in-memory state needs to be preserved
- The server just needs to know that it has an uncorrupted (not necessarily latest) version of the log.
- You can use this approach for both highly consistent and highly available systems (CAP)

Kafka is a log-oriented system that can be used to build other log-oriented systems

This just leaves one little problem…

How do you keep the log replicas up to date?

Primary-Backup Replication	Quorum-Based Replication
1 leader has the master copy and followers have backups	1 leader has the master copy and followers have backups
On WRITE, the master awaits the appending to **all** backup for acknowledging the client	On WRITE, the master waits on **only a majority** of the followers to confirm backups before it returns
Supports strong consistency for distributed READS, but doesn't scale easily and has lower throughput	Supports eventual consistency and higher throughput; doesn't require good networking between leader and followers
If the master is lost, restore from a backup	If a master is lost, elect a new leader from the replicas that have the latest data
F+1 replicas can tolerate F failures	2F+1 replicas can tolerate F failures

Kafka State Management

- Kafka brokers are stateless
  - They don’t track which messages a client has consumed or not.
  - This is the client’s job
  - Brokers simply send whatever the client requests
  - Compare to REST
- Brokers eventually must delete data
  - How does a broker know if all consumers have retrieved data?
  - It doesn’t. Kafka has a Service Level Agreement:
    - “Delete all data older than N days” for example

Kafka	Traditional Messaging: AMQP, JMS, Etc
Brokers are stateless. This makes broker distribution simpler. State management is done by producers and consumers.	Brokers are state-full. This makes distribution more difficult since load balancing and fault recovery are harder.
Messages can be delivered in batches	Messages are individually delivered
”At least once” delivery	“Exactly once” delivery
Eventual consistency model for messages partitioned across one broker	Strong consistency between brokers in a distributed system
Optimized for highly variable latency, large message throughput: streaming data, logs to late night batch pulls	Optimized for low latency delivery of smaller messages
Assumes replay of messages	Replay is an add-on

Apache Kafka resembles in some ways the REST architecture

Sources

- Kreps, J., Narkhede, N. and Rao, J., 2011, June. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB* (pp. 1-7).
- Wang, G., Koshy, J., Subramanian, S., Paramasivam, K., Zadeh, M., Narkhede, N., Rao, J., Kreps, J. and Stein, J., 2015. Building a replicated logging system with Apache Kafka. *Proceedings of the VLDB Endowment*, *8*(12), pp.1654-1655.
- https://kafka.apache.org/documentation/
- https://sookocheff.com/post/kafka/kafka-in-a-nutshell/

Log Processing and Distributed Messaging

- Companies like LinkedIn have both real-time and batch processing needs
- Batch processing: traditional business analytics
  - Done on Hadoop hourly or daily
  - Too slow to capture social network data: likes, views, updates to news feeds, etc
- Real-time processing needs to live side by side with batch processing
- They built a system that could do both: Apache Kafka