15-440 Distributed Systems, Fall 2021

Publish-Subscribe

Adapted from slides by Heather Miller

Carnegie Mellon University
School of Computer Science

A QUICK ANNOUNCEMENT BEFORE WE START…

The Spring ‘22 version of 15-440 needs TAs! If you’re interested, please reach out to Prof. Satya (satya@cs.cmu.edu).

Roadmap

- Motivation
- How does PubSub fit in?
- Apache Kafka, and how it works

Recall Spark and MapReduce?

Review...

What was Spark and MapReduce good for?

Usually good when you have...

✓ Batch workloads (no fine-grained updates to shared state)

Recall Spark and MapReduce?

What if you wanted to combine multiple kinds data-intensive systems like these?

e.g.,

The Big-Data Ecosystem Table

> *Incomplete-but-useful list of big-data related projects packed into a JSON dataset.*
>
> - Github repository: https://github.com/zenkay/bigdata-ecosystem
> - Raw JSON data: http://bigdata.andreamostosi.name/data.json
> - Original page on my blog: http://blog.andreamostosi.name/big-data/
>
> by Andrea Mostosi (http://blog.andreamostosi.name)

Frameworks		
Apache Hadoop	framework for distributed processing. Integrates MapReduce (parallel processing), YARN (job scheduling) and HDFS (distributed file system)	1. Apache Hadoop
Distributed Programming		
AddThis Hydra	Hydra is a distributed data processing and storage system originally developed at AddThis. It ingests streams of data (think log files) and builds trees that are aggregates, summaries, or transformations of the data. These trees can be used by humans to explore (tiny queries), as part of a machine learning pipeline (big queries), or to support live consoles on websites (lots of queries).	1. Github
Akela	Mozilla's utility library for Hadoop, HBase, Pig, etc.	1. Website
Amazon Lambda	a compute service that runs your code in response to events and automatically manages the compute resources for you	1. Website
Amazon SPICE	Super-fast Parallel In-memory Calculation Engine	1. Website
AMPcrowd	A RESTful web service that runs microtasks across multiple crowds	1. Website
AMPLab G-OLA	a novel mini-batch execution model that generalizes OLA to support general OLAP queries with arbitrarily nested aggregates using efficient delta maintenance techniques	1. Website
AMPLab SIMR	Apache Spark was developed thinking in Apache YARN. However, up to now, it has been relatively hard to run Apache Spark on Hadoop MapReduce v1 clusters, i.e. clusters that do not have YARN installed. Typically, users would have to get permission to install Spark/Scala on some subset of the machines, a process that could be time consuming. SIMR allows anyone with access to a Hadoop MapReduce v1 cluster to run Spark out of the box. A user can run Spark directly on top of Hadoop MapReduce v1 without any administrative rights, and without having Spark or Scala installed on any of the nodes.	1. SIMR on GitHub
Apache Crunch	is a simple Java API for tasks like joining and data aggregation that are tedious to implement on plain MapReduce. The APIs are especially useful when processing data that does not fit naturally into relational model, such as time series, serialized object formats like protocol buffers or Avro records, and HBase rows and columns. For Scala users, there is the Scrunch API, which is built on top of the Java APIs and includes a REPL (read-eval-print loop) for creating MapReduce pipelines.	1. Website
Apache DataFu	DataFu provides a collection of Hadoop MapReduce jobs and functions in higher level languages based on it to perform data analysis. It provides functions for common statistics tasks (e.g. quantiles, sampling), PageRank, stream sessionization, and set and bag operations. DataFu also provides Hadoop jobs for	1. DataFu Apache Incubator 2. LinkedIn DataFu

There are lots of different kinds of big data systems out there

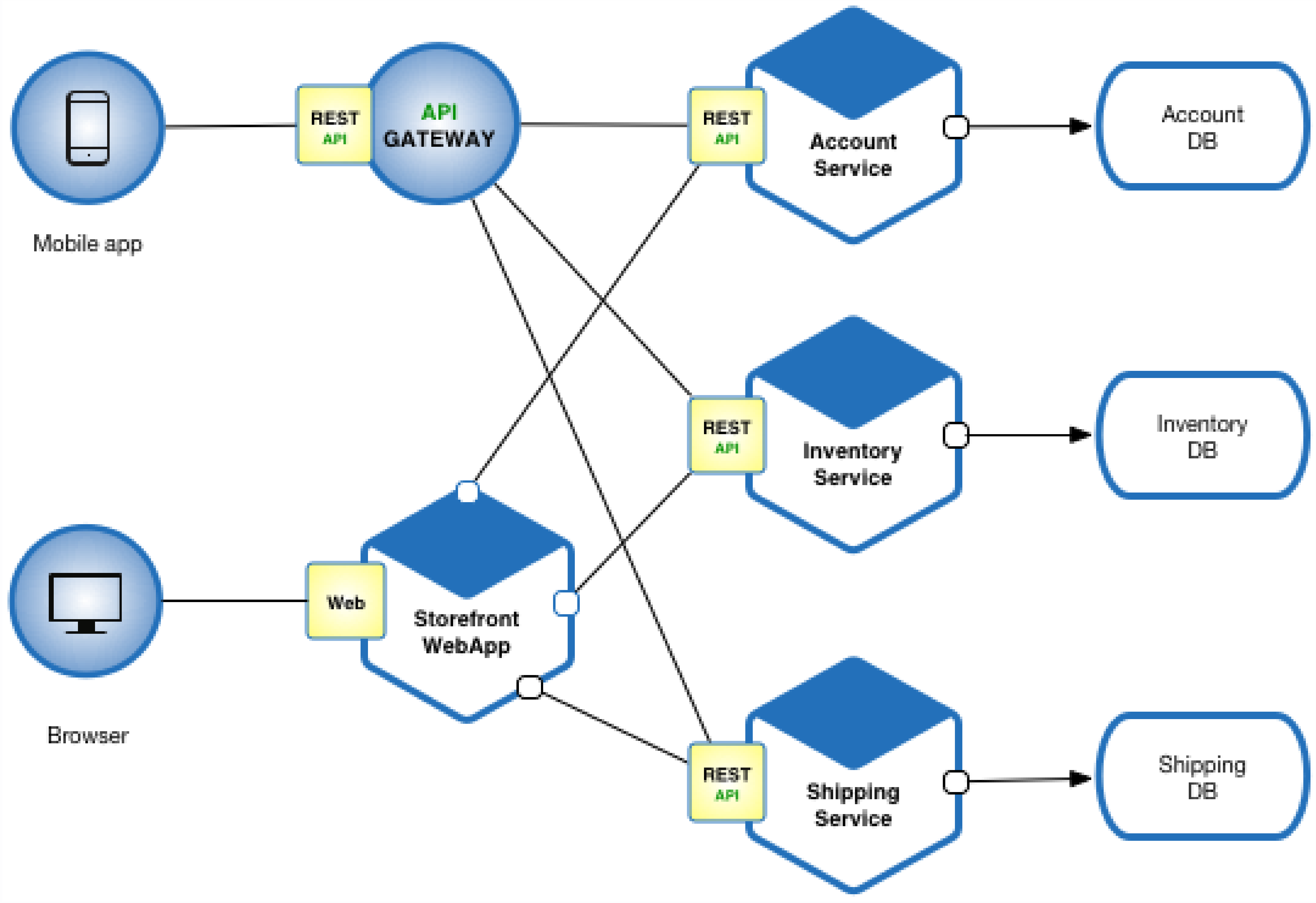
BEYOND BIG-DATA SYSTEMS…

Today, popular web applications are often built up of hundreds of small, communicating components

THE HIGH LEVEL IDEA:

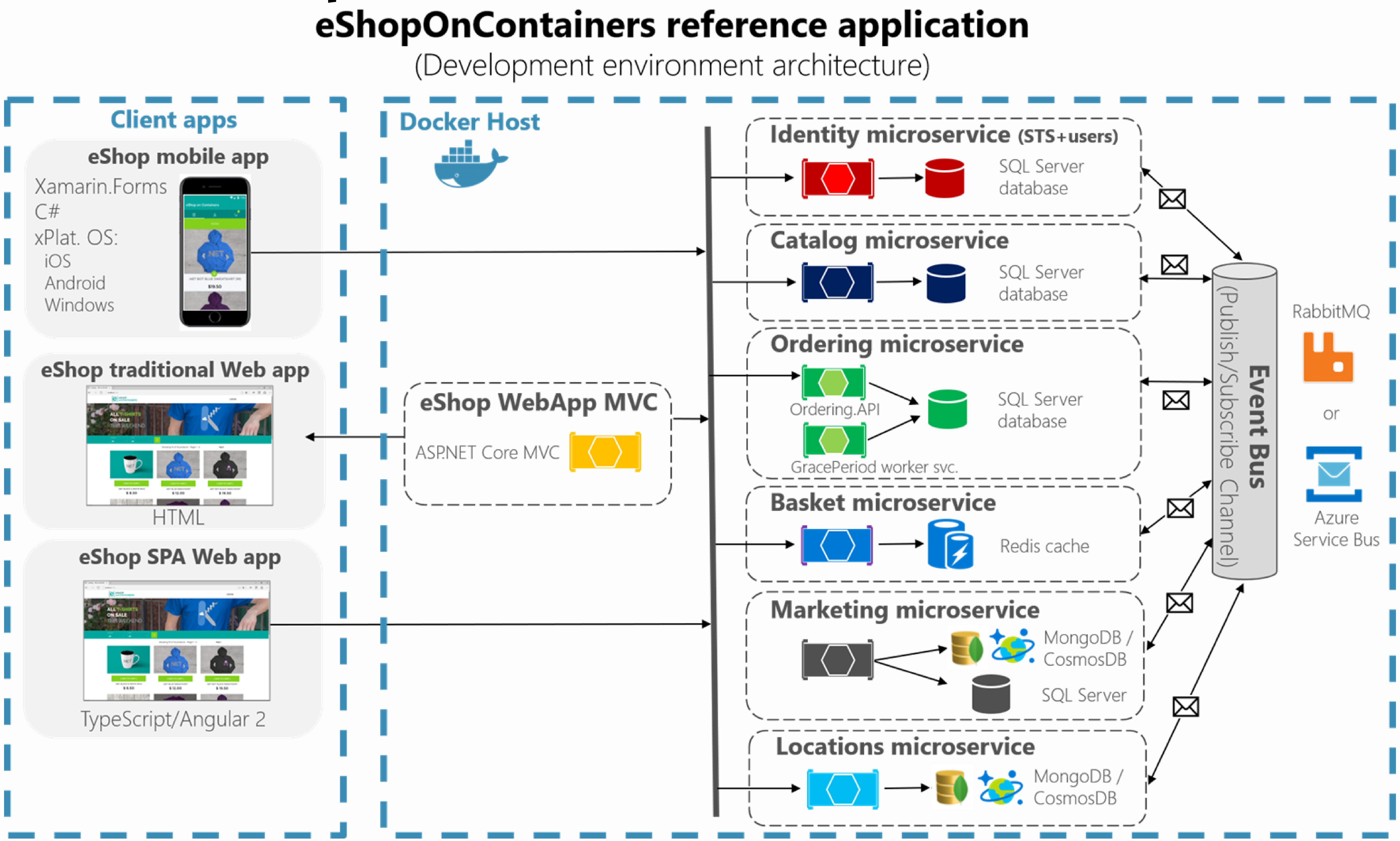
Microservice Philosophy

Application is a collection of small, modular, replaceable, independently deployable “services”.



EXAMPLE:

Mobile Shop



SOURCE:https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/multi-container-microservice-net-applications/microservice-application-design

Class Exercise

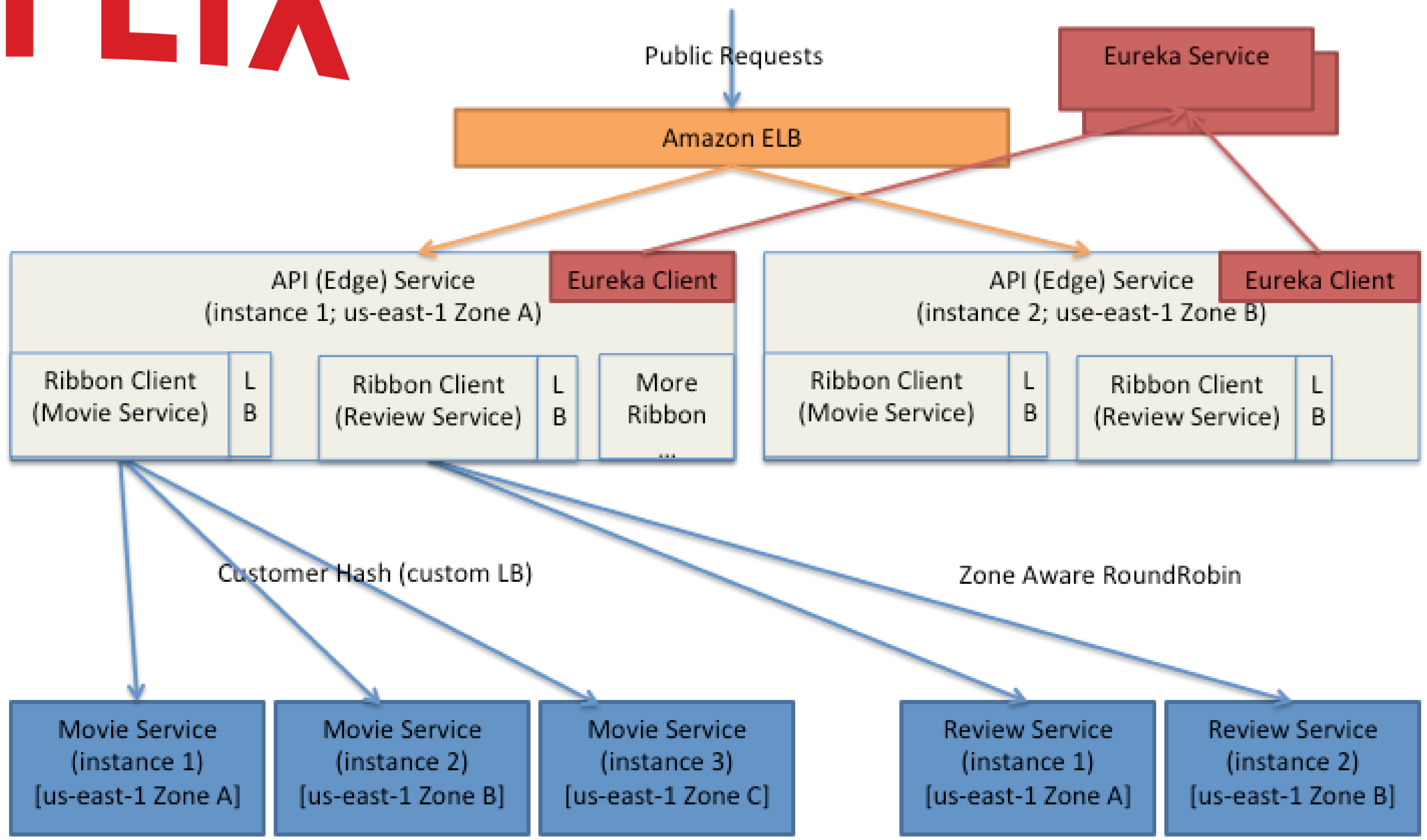
What types of microservices would you expect Netflix to run in production?

Class Exercise

NETFLIX

[Services Netflix has:]

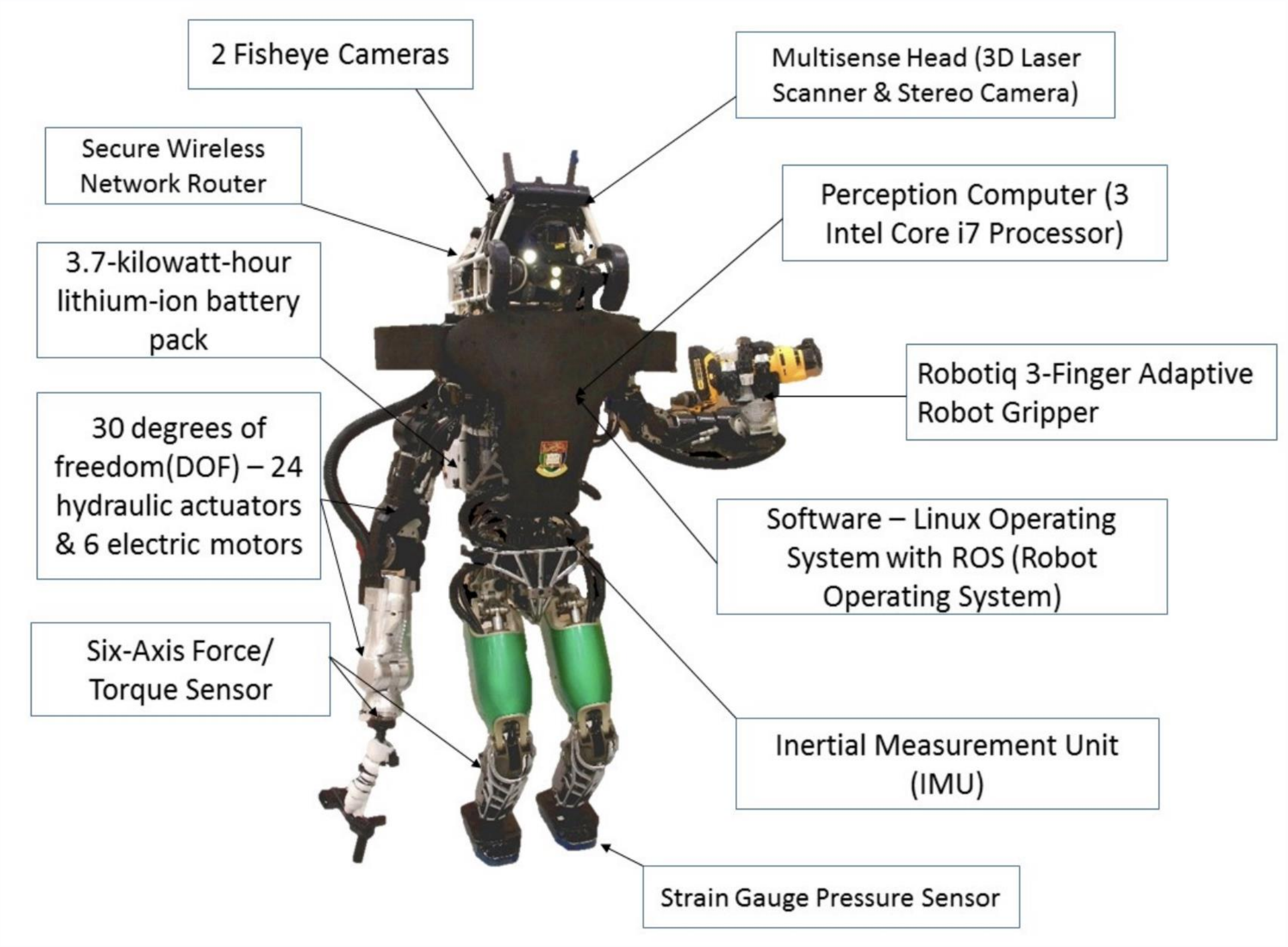
- API Gateway Service
- Titles Service (for Metadata)
- Ratings Service
- Movies Service (for Data)
- Accounts Service
- A/B Testing Service
- Service Failure Service
- Rescheduling Service
- Log Collection Service
- Service Status Visualization Service
- Transaction Service
- ...



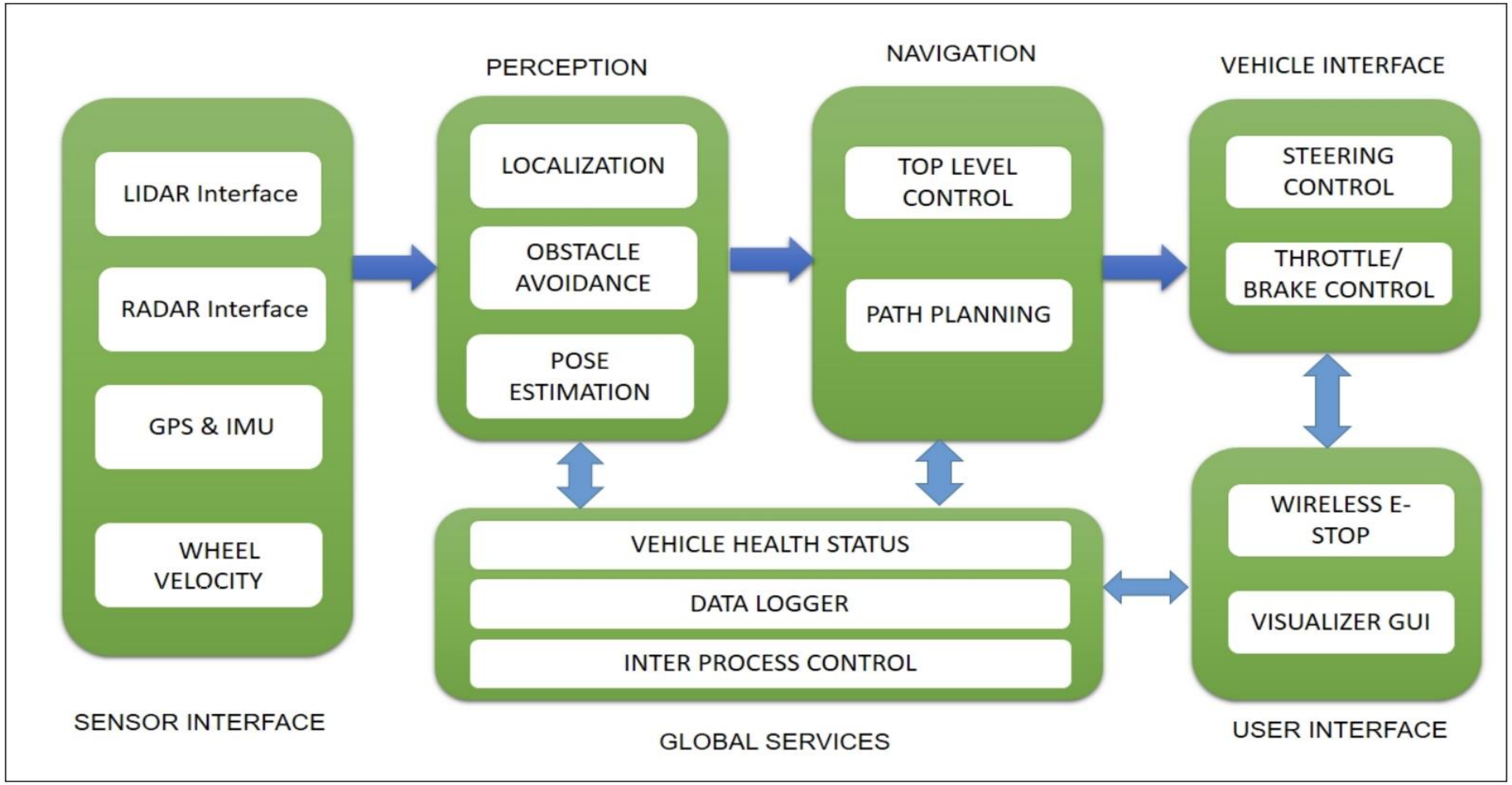
BEYOND WEB APPS…

Robots can be thought of as distributed systems of communicating components too!

Robots are distributed systems!



Robots are distributed systems!



Robot Operating System (ROS)

HOW SHOULD THEY ALL COMMUNICATE?

RPCs?

HOW SHOULD THEY ALL COMMUNICATE?

Two issues with RPCs:

1. Synchronization (both components need to be alive at the same time)

2. Tight coupling between components (components need to know about each other)

HOW SHOULD THEY ALL COMMUNICATE?

RPC is too hands-on for managing *all* of these components, and their replicas!

ENTER:

Publish-Subscribe

Gist:

A way for the different parts of a system to communicate with each other

Each component (i.e. node) can:

- **Publish**: send messages regardless of who is listening
- **Subscribe**: receive messages regardless of who is sending

THE HIGH LEVEL IDEA:

Publish-Subscribe

Basic idea:

A model in which we provide a framework to glue requestors (**producers**) to workers (**consumers**), with much looser coupling.

On the producer side:

Requests are made as **published messages** on **topics**.

On the consumer side:

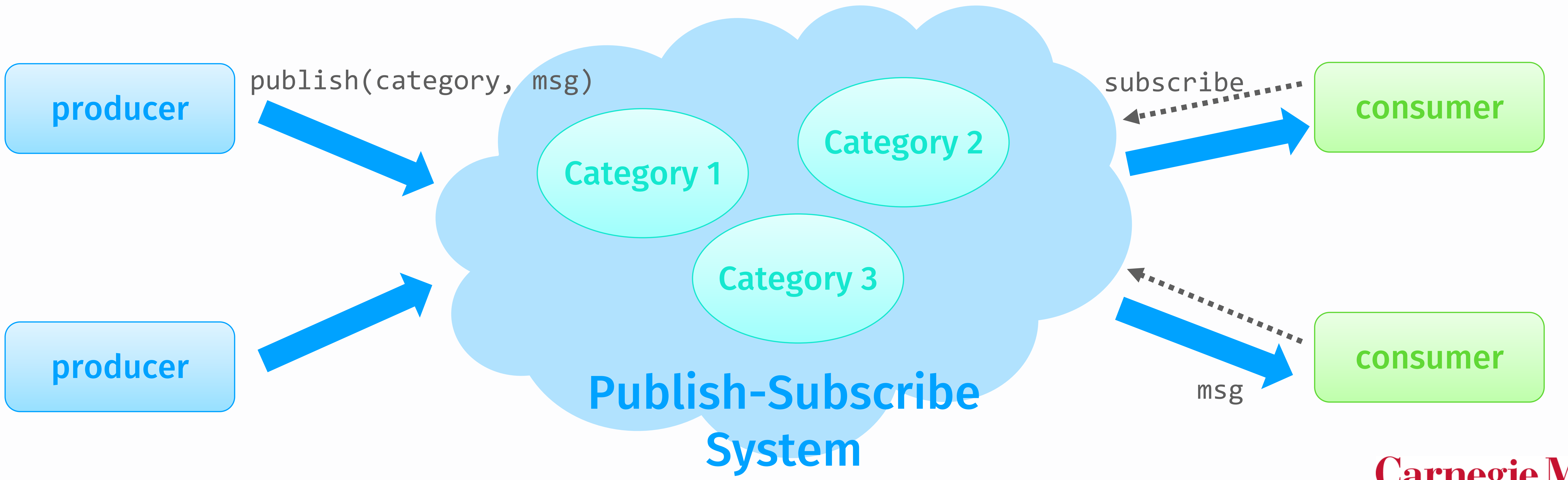
Workers monitor topics (**subscribe**) and then an idle worker can announce that it has taken on some task, and later, finished it.

THE HIGH LEVEL IDEA:

Publish-Subscribe

Basic idea:

A model in which we provide a framework to glue requestors (**producers**) to workers (**consumers**), with much looser coupling.



Subscription Models

Topic-based:

Events are classified into predefined topics. Subscriptions can include any number of these topics.

E.g.,

- *International Film Festivals in Pittsburgh*
- *Weather in Pittsburgh*

Content-based:

Events are structured in the form of multiple attributes. Subscriptions can define a range over any of these attributes.

E.g.,

- *Temperature between 25F and 40F*

A popular publish-subscribe framework:

Franz Kafka
Bohemian novelist

A popular publish-subscribe framework:

APACHE kafka®

Can be thought of as a **distributed** publish-subscribe messaging system.

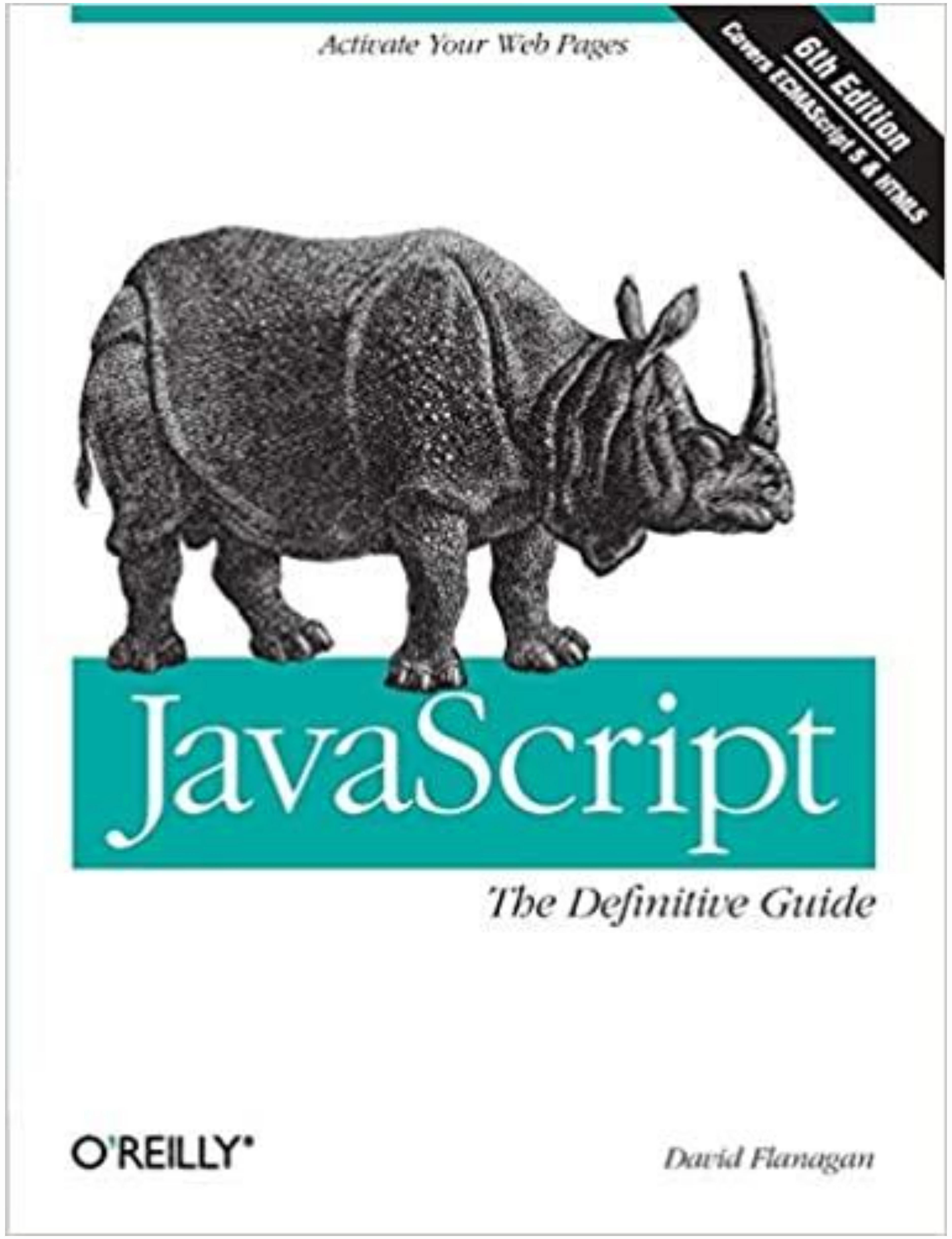
Features:

- High Availability
- High Throughput
- Scalability
- Durability (message still received, even if queue is offline)

Apache Kafka

*"The main value Kafka provides to data pipelines is its ability to serve as a **very large, reliable buffer** between various stages in the pipeline, effectively decoupling producers and consumers of data within the pipeline.*

This decoupling, combined with reliability, security, and efficiency, makes Kafka a good fit for most data pipelines."


Activate Your Web Pages
6th Edition
Covers ECMAScript 5 & HTML5
JavaScript
The Definitive Guide
O'REILLY
David Flanagan


???
???

Apache Kafka

*"The main value Kafka provides to data pipelines is its ability to serve as a **very large, reliable buffer** between various stages in the pipeline, effectively decoupling producers and consumers of data within the pipeline.*

This decoupling, combined with reliability, security, and efficiency, makes Kafka a good fit for most data pipelines."

Kafka Design Goals

Built at LinkedIn

Motivation:

"A unified platform for handling all the real-time data feeds a large company might have."

Must haves:

- *High throughput* to support high volume event feeds.
- Support *real-time processing* of these feeds to create new, derived feeds.
- *Support large data backlogs* to handle periodic ingestion from offline systems.
- Support *low-latency* delivery to handle more traditional messaging use cases.
- Guarantee *fault-tolerance* in the presence of machine failures.

Kafka at LinkedIn (2014)

What type of data is being transported through Kafka?

- Metrics: operational telemetry data.
- Tracking: everything a LinkedIn user does.
- Queuing: between LinkedIn apps, e.g., for sending emails.

Used to transport data from LinkedIn's apps to Hadoop and back

- In total ~200 billion events/day via Kafka
- Tens of thousands of data producers, thousands of consumers
- 7 million events/sec (write), 35 million events/sec (read)
  > Many replicated events

Multiple clusters across multiple data centers

Core concepts in Kafka

- **Records** have a **key (optional), value**, and **timestamp**
- **Topic** is a stream of records (e.g., orders)
- **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.

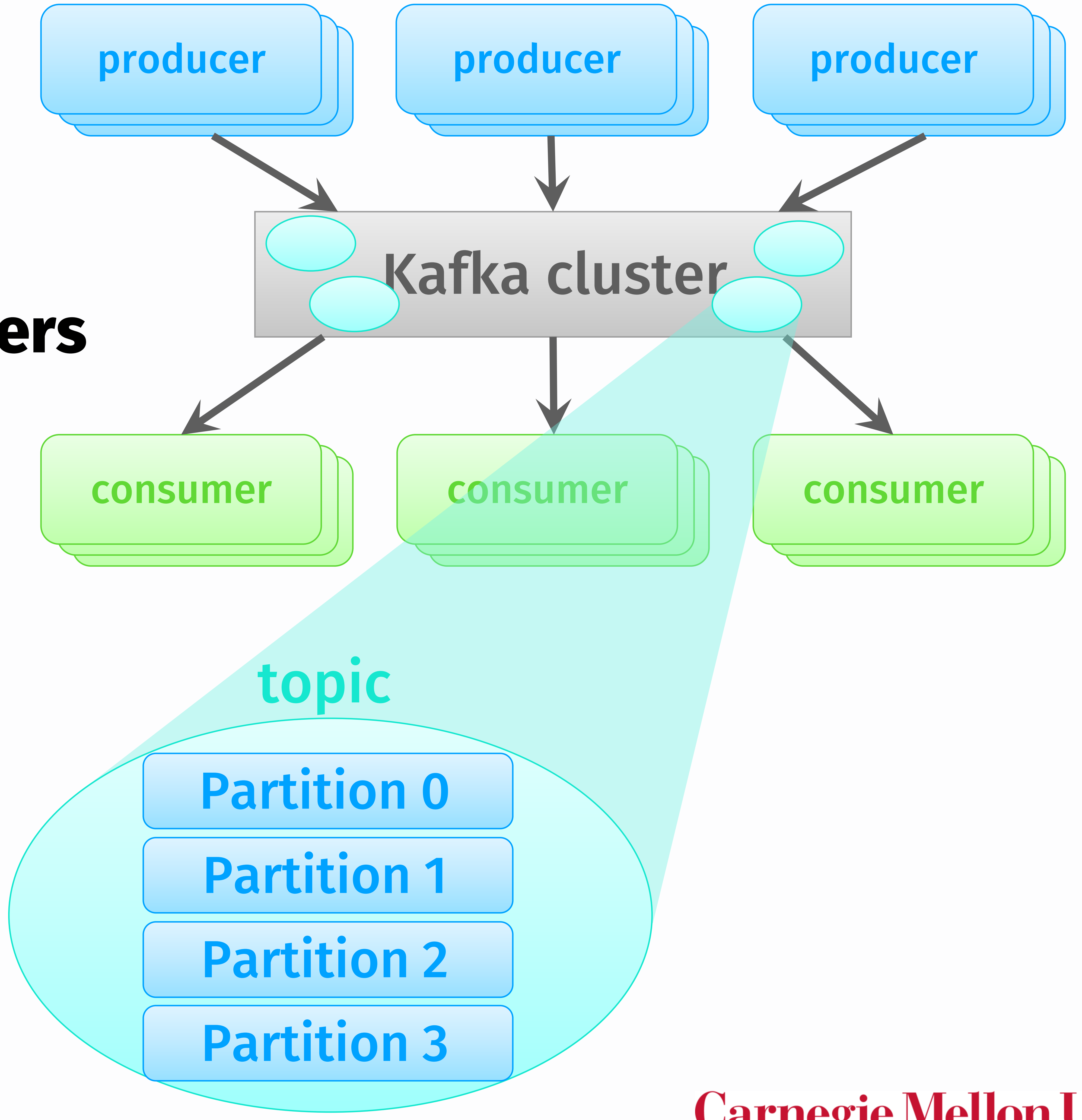
Kafka at a glance

Who does what

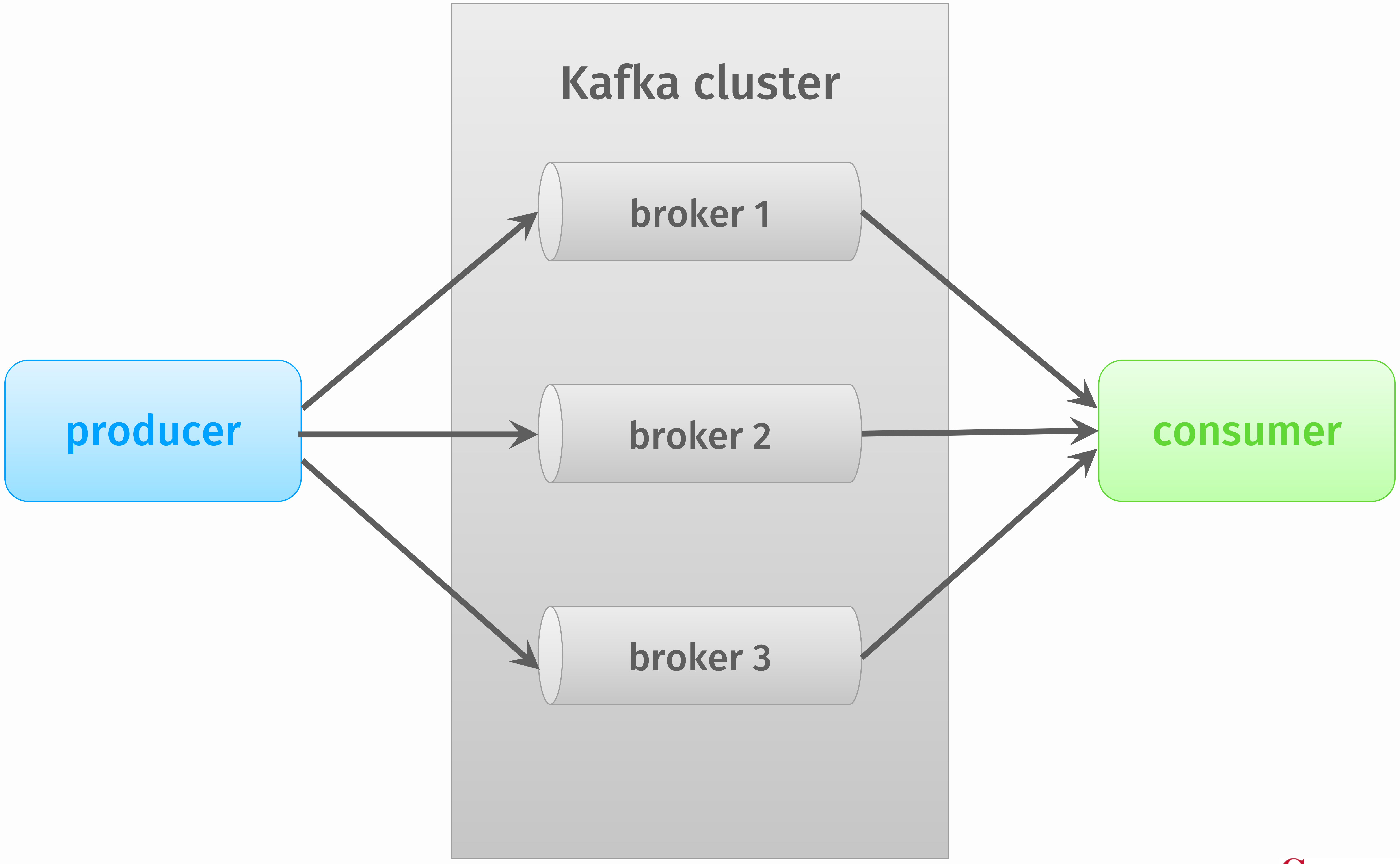
- **Producers** write data to **brokers**
- **Consumers** read data from **brokers**
- All of this is distributed

The data

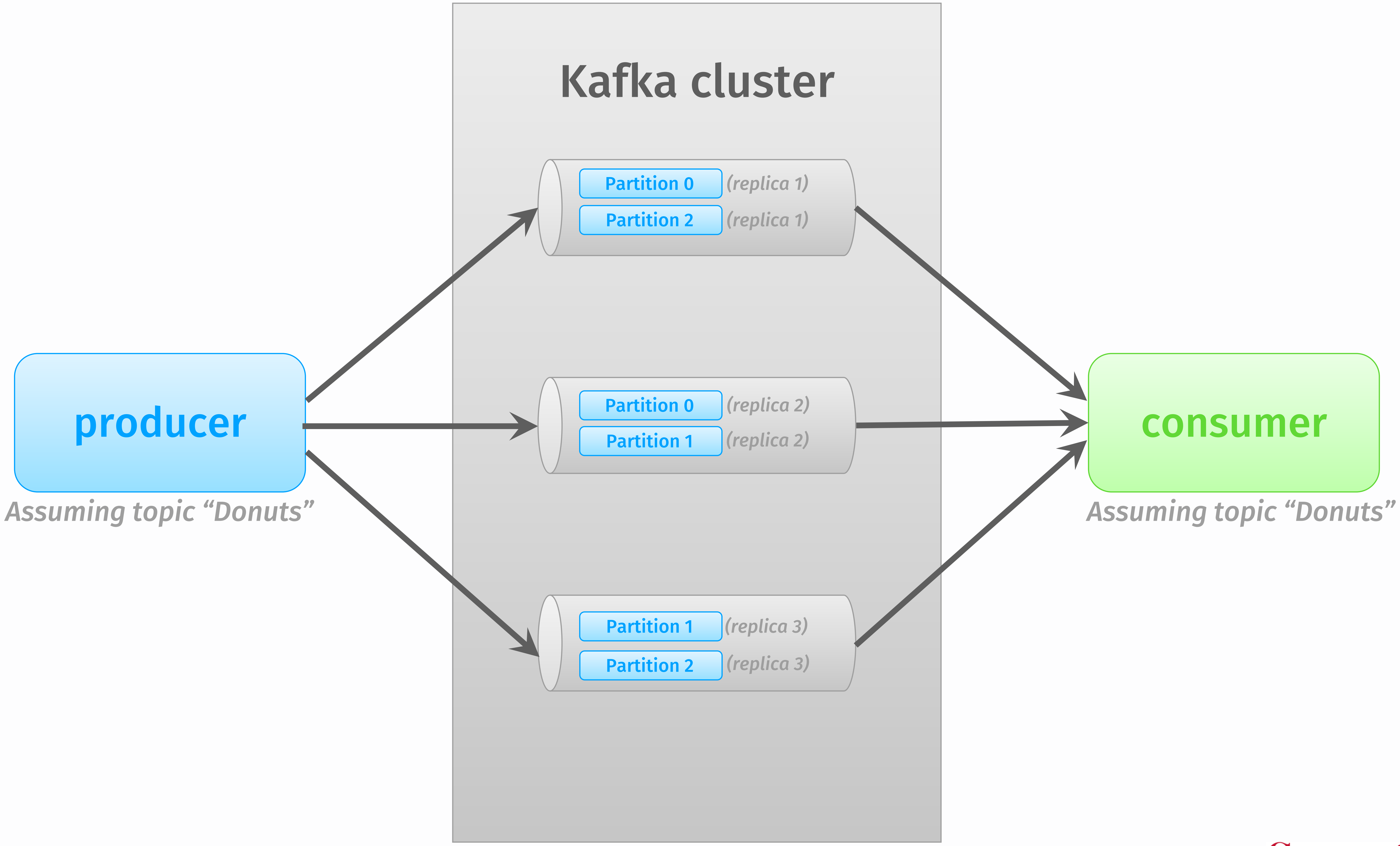
- Data is stored in **topics**
- **Topics** are split into **partitions** which are **replicated**



Core concepts in Kafka



Core concepts in Kafka



Carnegie Mellon University
School of Computer Science

Topics

Let's first dive into the core abstraction Kafka provides for a stream of records — the topic.

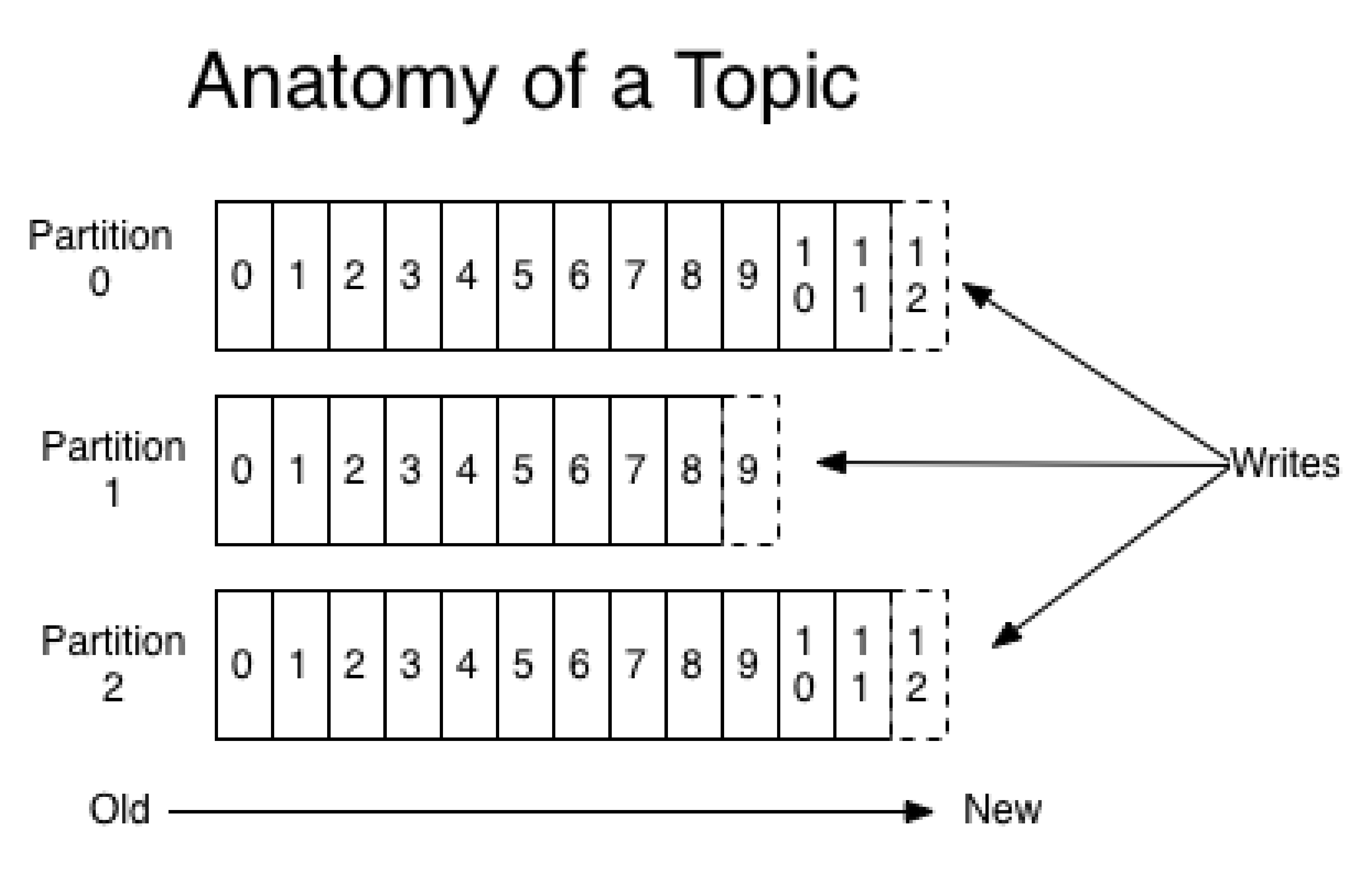
A topic is a *category* or *feed name* to which records are published (think of it like a **label**)

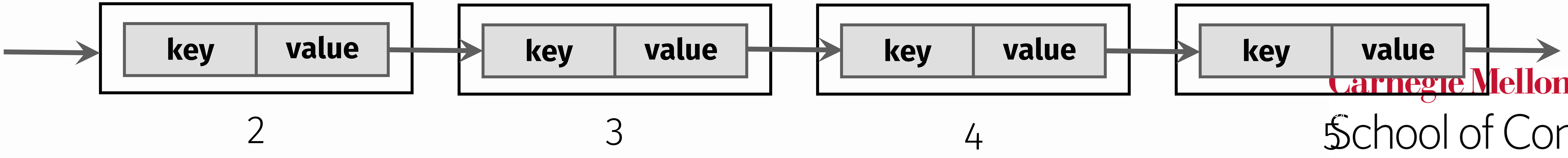
A topic can have zero, one, or many consumers that subscribe to the data written to it.

Partitions

A topic consists of a configurable number of **partitions**.

Partition:
ordered + immutable
sequence of messages that is continually appended to





Partitions

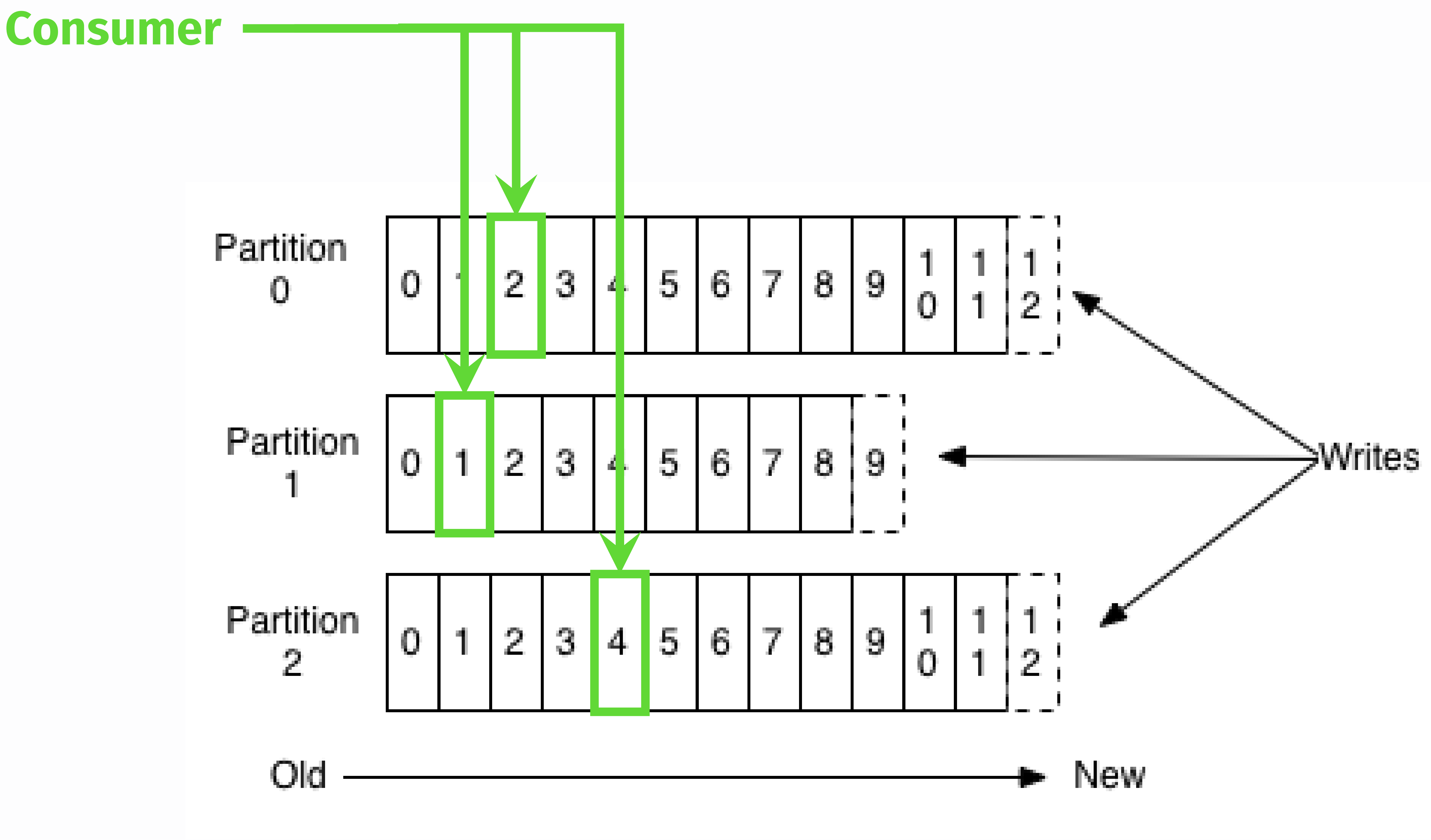
Partitions exist to improve *performance*

- More #Partitions → Higher consumer parallelism (later)

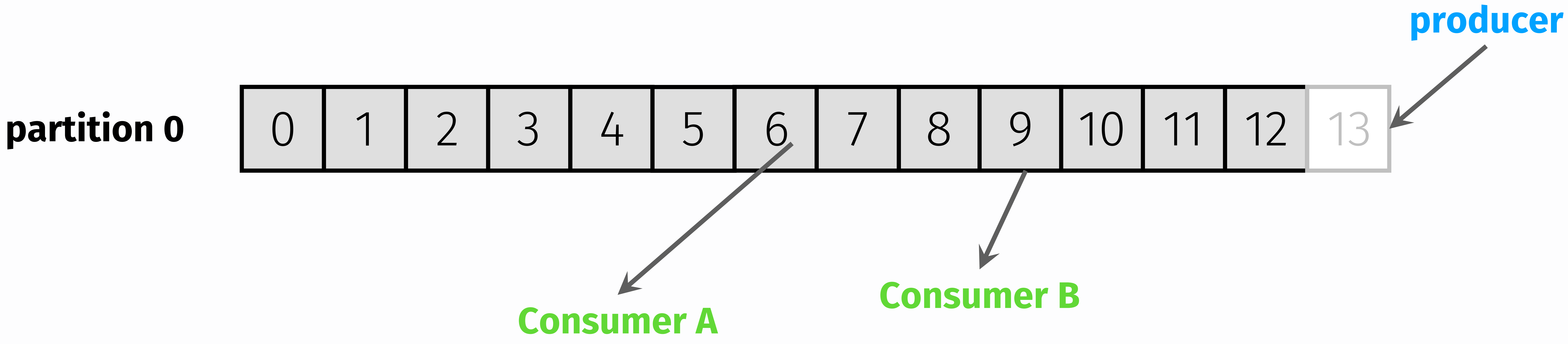
Partition offsets

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



Consumers each have their own offset.

Producer writing to offset 13 of partition 0 while...
Consumer A is reading from offset 6.
Consumer B is reading from offset 9.

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.
- 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

Topic

Label for the data

↓

Partition

Increases consumer parallelism

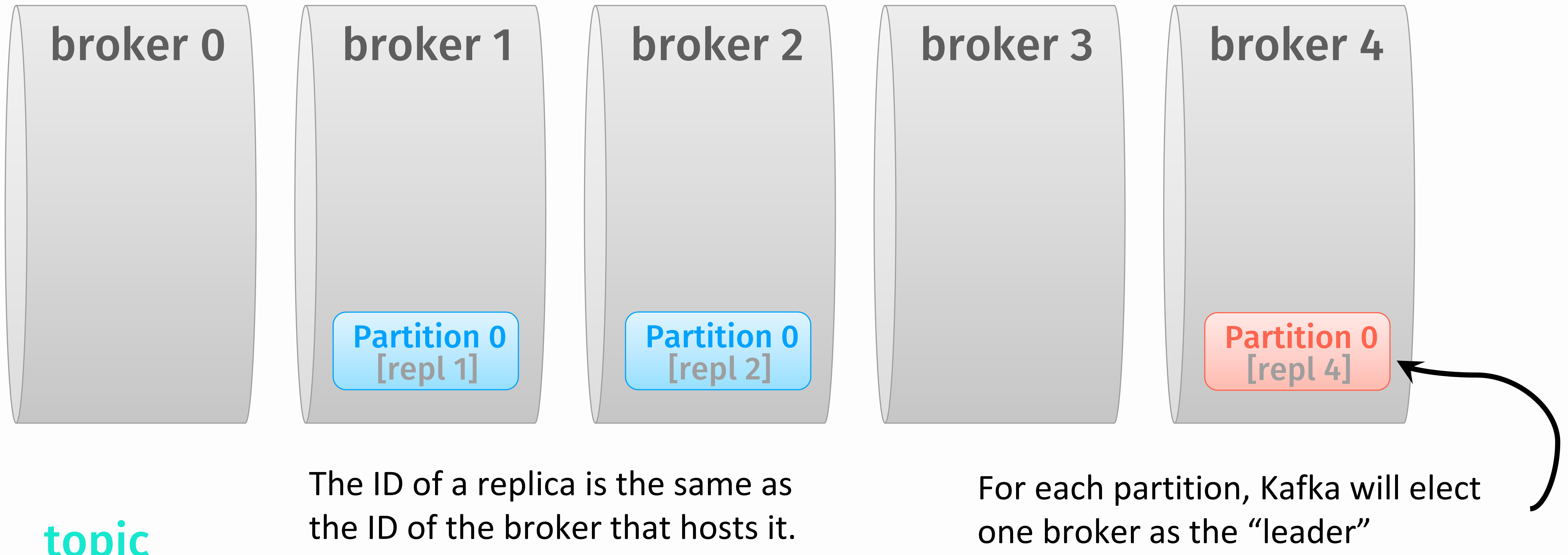
→

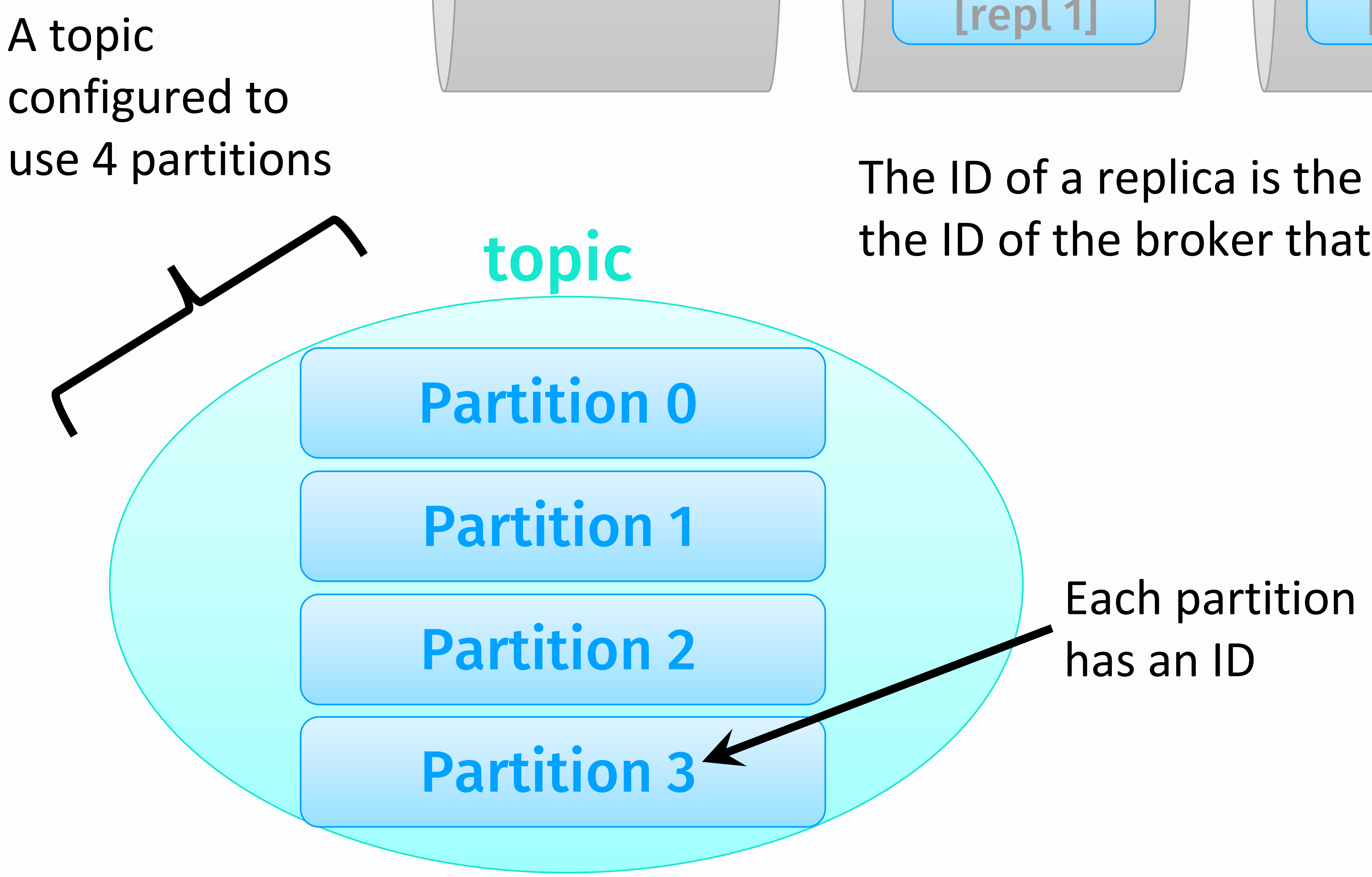
Replica

Copy of partition, fault-tolerance

Topics vs Partitions vs Replicas

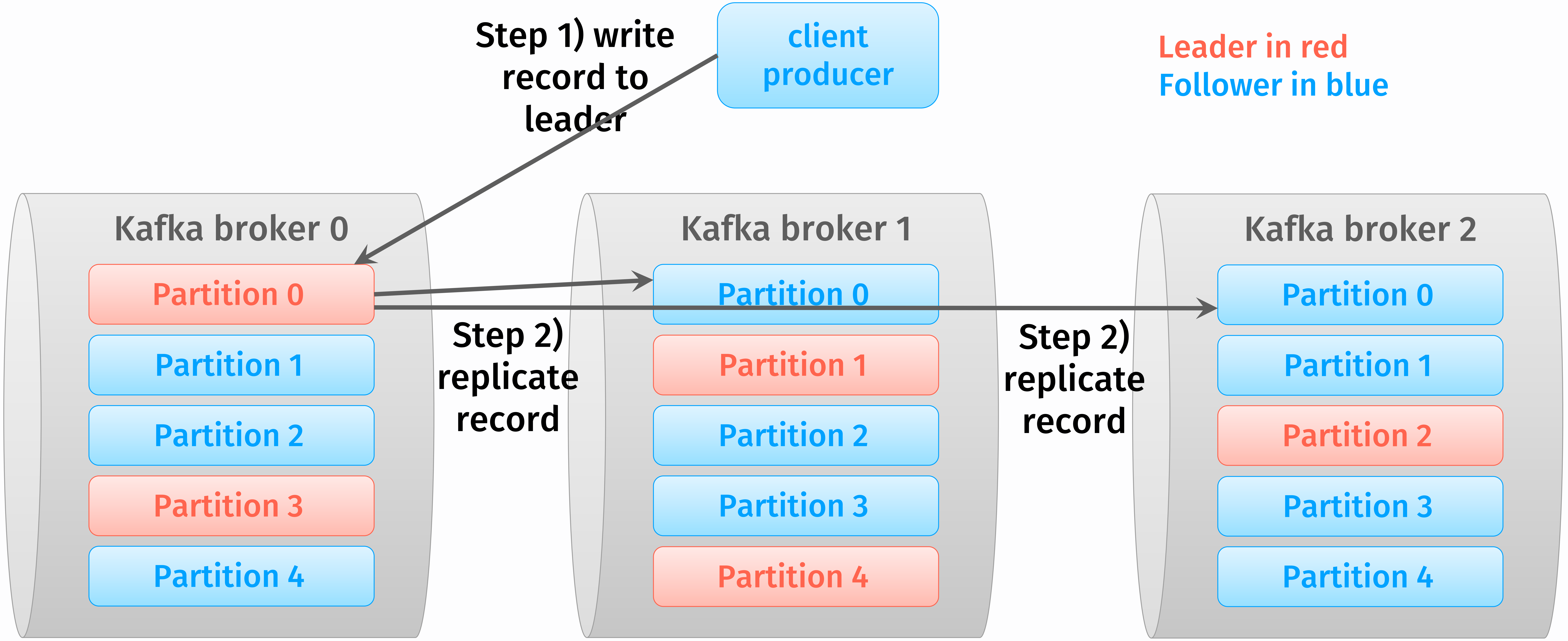
Leader in red
Follower in blue



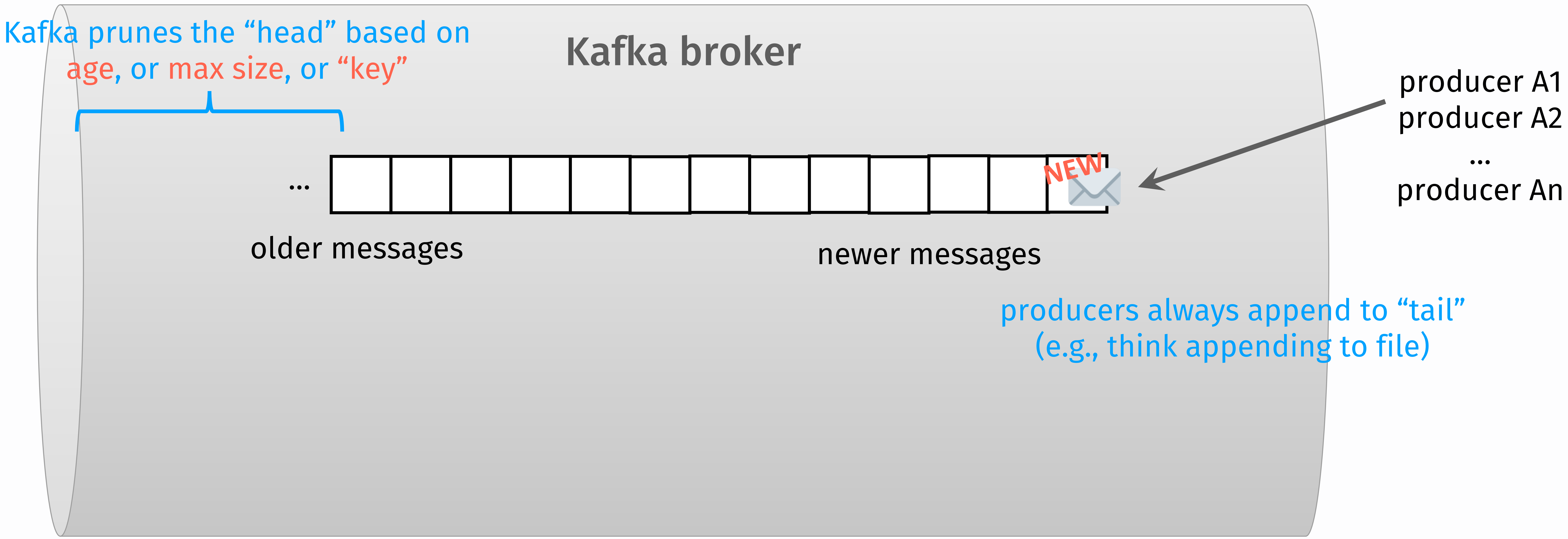


If, say, the replication factor of a topic is set to 3, then Kafka will create 3 identical replicas of each partition and place those on available brokers in the cluster.

Propagating writes across replicas



Dealing with Finite Storage



Kafka Producers

- Producers send records to topics
- Producer picks which partition to send record to per topic
  - Can be done **round-robin**
  - Can be based on priority
  - Typically based on **key** of **record**

Remember! Producer picks partition.

Kafka Producers

- Producers write at their own cadence so order of records cannot be guaranteed across partitions.
- Producer configures consistency level (ack=0, ack=all, ack=1).
- Producers pick the partition such that records/messages go to a given same partition based on the data (usually key).
  - *Example: have all the events of a certain EmployeeID go to the same partition.*
  - *If order within a partition is not needed, a round-robin partition strategy can be used so records are evenly distributed across partitions.*

Ways to send messages

Fire-and-forget

We send a message to the server and don't really care if it arrives successfully or not. Some messages will get lost using this method.

Synchronous send

We send a message, the send() method returns a Future object, and we use get() to wait on the future and see if the send() was successful or not.

Asynchronous send

We call the send() method with a callback function, which gets triggered when it receives a response from the Kafka broker.

Putting Asynchrony into Context

Suppose the network roundtrip time between our application and the Kafka cluster is 10ms.

If we wait for a reply after sending each message, sending 100 messages will take ~1 second. **(Synchronous)**

On the other hand, if we just send all our messages and not wait for any replies, then sending 100 messages will barely take any time at all. **(Fire-and-Forget)**

On the other hand, we may need to know when we failed to send a message completely so we can throw an exception or log an error. For this purpose, Kafka supports producer callbacks. **(Asynchronous)**

Parameters affecting Producer Performance

Two aspects worth mentioning because they significantly influence Kafka performance:

1. Message ACKing
2. Message Batching

1) Message ACKing

In Kafka, a message is considered *committed* when **any required** ISR (in-sync replicas) for that partition have applied it to the data log.

- Message ACKing is about conveying this “Yes, committed!” information back to the producer from the data brokers.
- Exact meaning of **any required** depends on chosen semantics

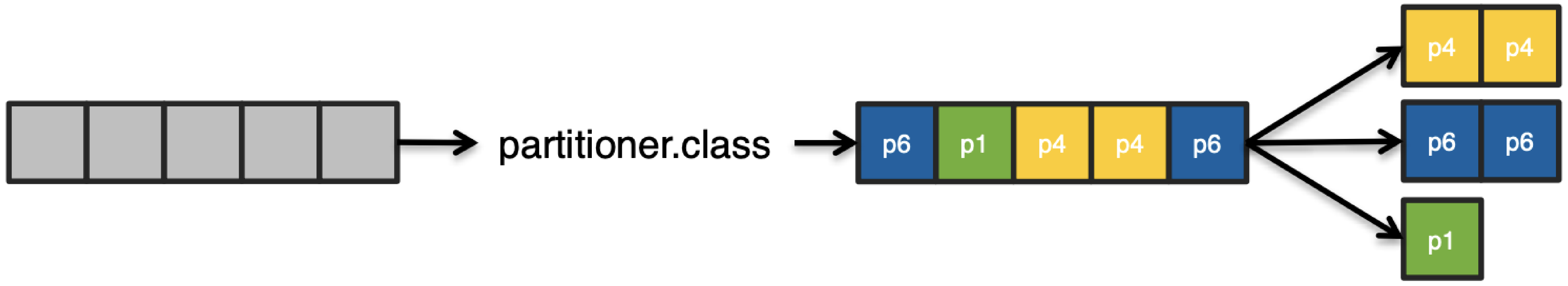
1) Message ACKing

better latency ↕ better durability

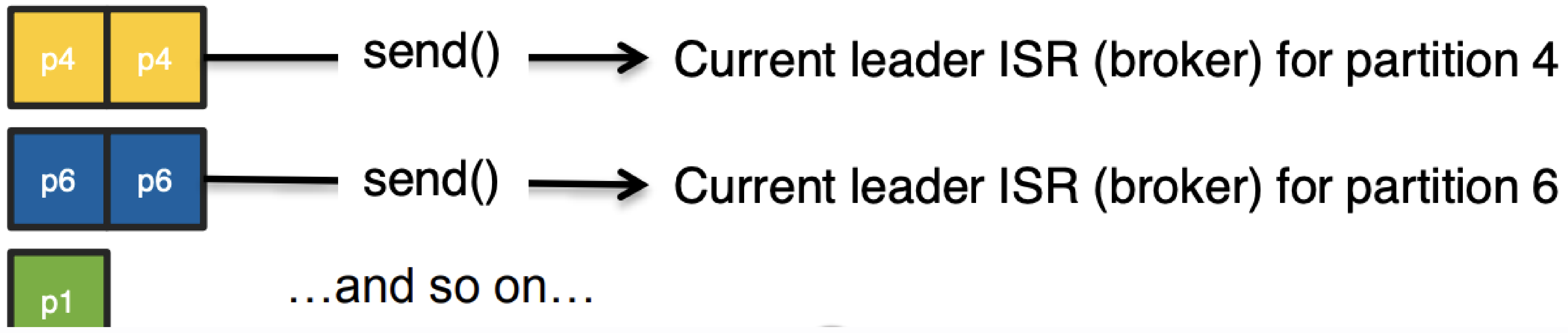
- Typical values of `request.required.acks`
  - `0`: producer never waits for an ack from the broker.
    - Gives **the lowest latency** but the weakest durability guarantees.
  - `1`: producer gets an ack after the leader replica has received the data.
    - Gives better durability as the we wait until the lead broker acks the request. Only msgs that were written to the now-dead leader but not yet replicated will be lost.
  - `all`: producer gets an ack after *all* ISR have received the data.
    - Gives **the best durability** as Kafka guarantees that no data will be lost as long as at least one ISR remains.

2) Message Batching

- Batching improves throughput
  - Tradeoff is data loss if client dies before pending messages have been sent.
  - The original list of messages is partitioned (randomly if the default partitioner is used) based on their destination partitions/topics, i.e. split into smaller batches.



  - Each post-split batch is sent to the respective leader broker/ISR (the individual `send()`'s happen sequentially), and each is acked by its respective leader broker according to `request.required.acks`.



Kafka Consumers

Consumers *pull* from Kafka (there's no *push*)

- Allows consumers to control their pace of consumption
- Allows to design downstream apps for **average** load instead of **peak** load

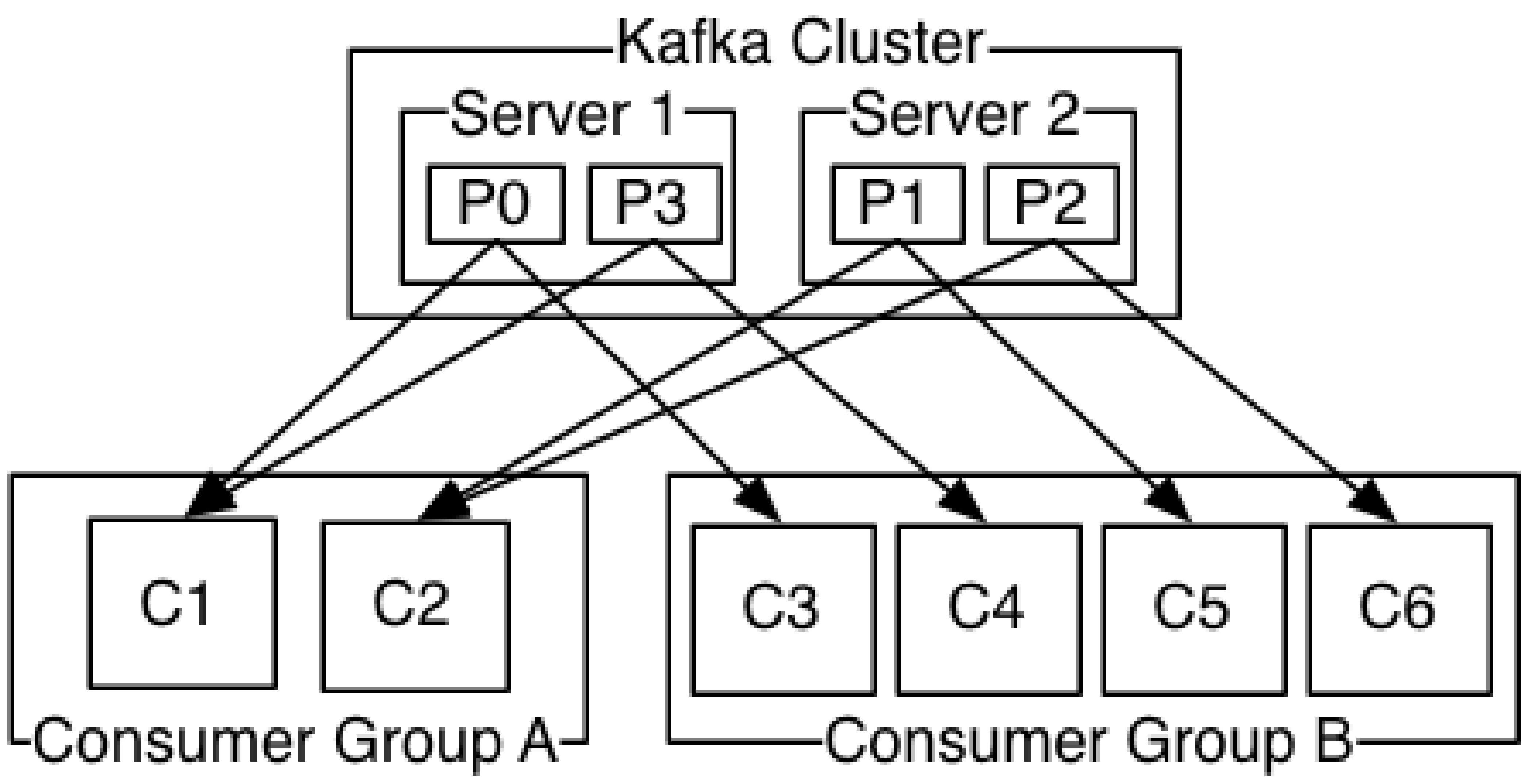
Consumers are responsible for tracking their read positions (aka "offsets").
What does offset management allow you to do?

- Consumers can **rewind in time** (up to the point where Kafka prunes), *e.g.* to replay older messages
- Consumers can decide to read a **subset of partitions** for a specific topic
- **Run offline**, periodically fetch batch updates

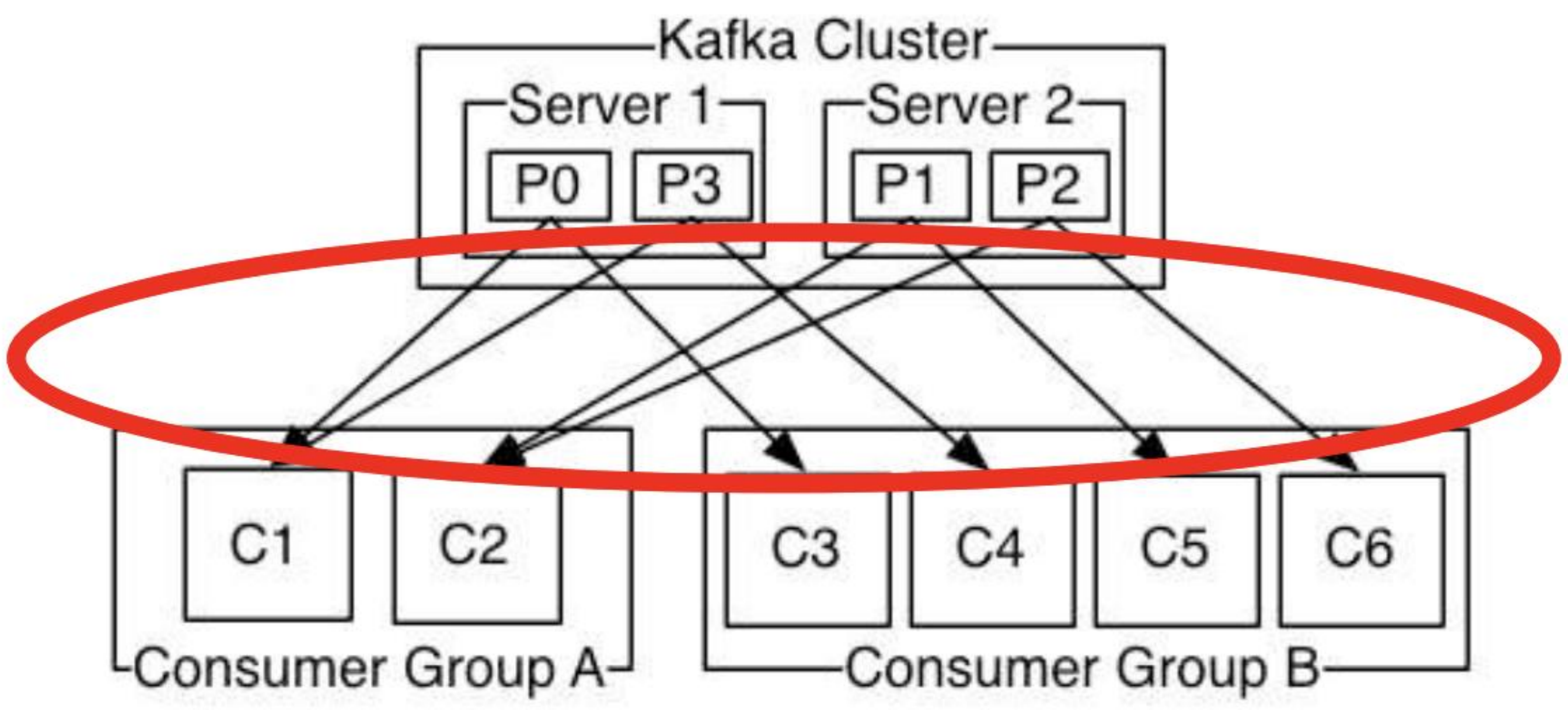
Kafka Consumers

Consumer Groups

- Allows multi-threaded/multi-machine consumption from Kafka topics
- Consumers “join” a group by using the same *group id*
- Kafka guarantees that a record is only ever read by one consumer in a group
  > Each partition is consumed by exactly one consumer in the group
  > Maximum parallelism: #consumers in group ≤ #partitions



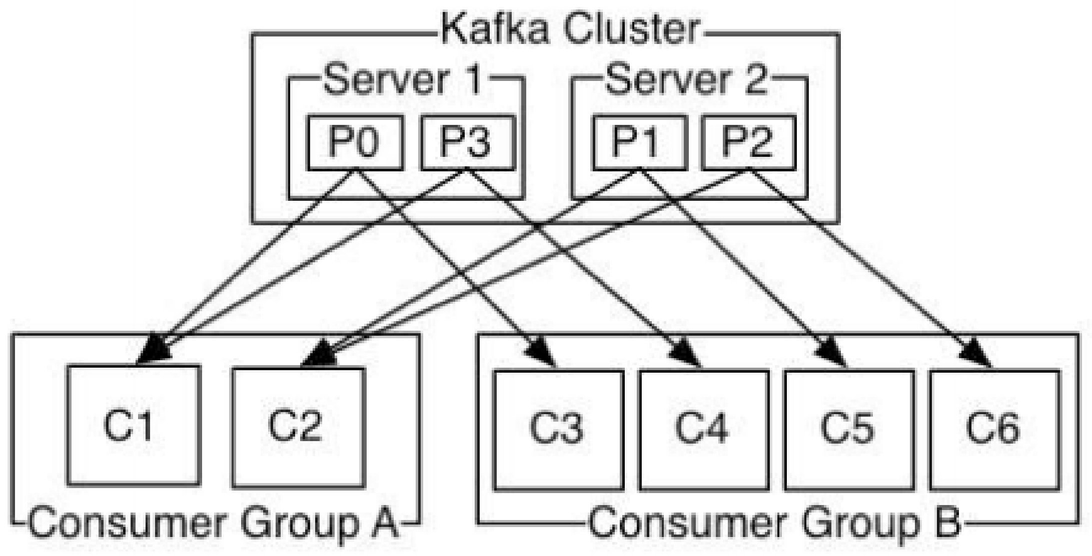
Rebalancing: how consumers meet brokers



The assignment of brokers – via the partitions of a topic – to consumers is quite **important**, and it is **dynamic** at run-time.

Rebalancing: how consumers meet brokers

- Why "dynamic at run-time"?
  - Machines can die, be added, …
  - Consumer apps may die, be re-configured, added, …



Rebalancing: how consumers meet brokers

- **Rebalancing?**
  - Consumers in a group come into consensus on which consumer is consuming which partitions → required for distributed consumption
  - Divides broker partitions evenly across consumers, tries to reduce the number of broker nodes each consumer has to connect to
- When does it happen? Each time:
  - a **consumer** joins or leaves a consumer group, OR
  - a **broker** joins or leaves, OR
  - a topic "joins/leaves" via a filter, cf. `createMessageStreamsByFilter()`
- Examples:
  - If a consumer or broker fails to heartbeat to ZK → rebalance!
  - `createMessageStreams()` registers consumers for a topic, which results in a rebalance of the consumer-broker assignment.

Other widely-used pubsub frameworks

Google Cloud Pub/Sub

Apache Kafka

Amazon Simple Notification Service

Redis

ROS (Robot Operating System)

Recap

Publish-Subscribe (PubSub)
- "Glue" framework between disparate components in a distributed system
- Unlike RPCs, allows components to remain **loosely-coupled**

Apache Kafka (Developed at LinkedIn in 2011)
- Abstraction: "very large, reliable buffer" of **topic-based** data
- **Producers**: Write records (key-value pairs) into an append-only log
- **Brokers** form Kafka cluster, manage replicated partitions within each topic
- **Consumers**: Read records from partitions, using *Consumer Groups* for parallelism