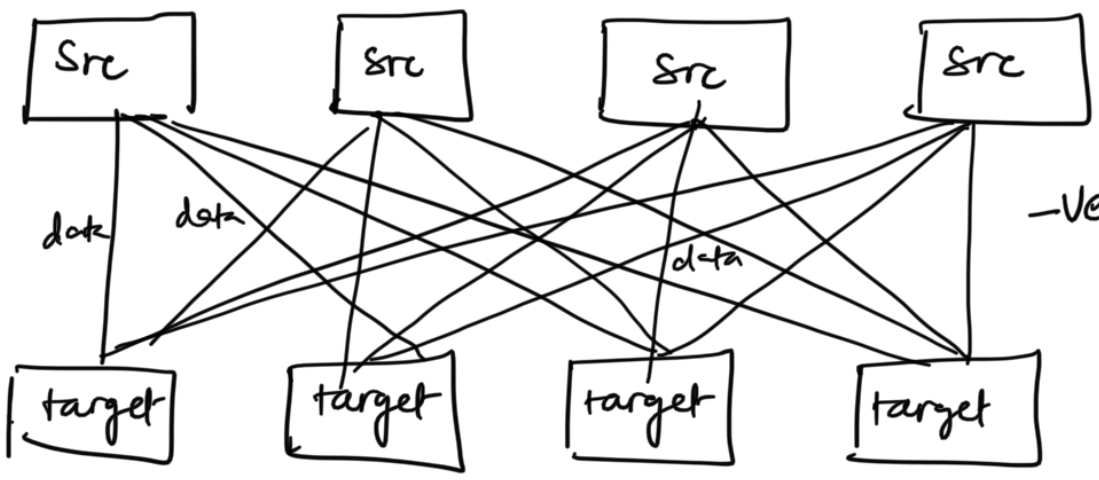
Apache Kafka

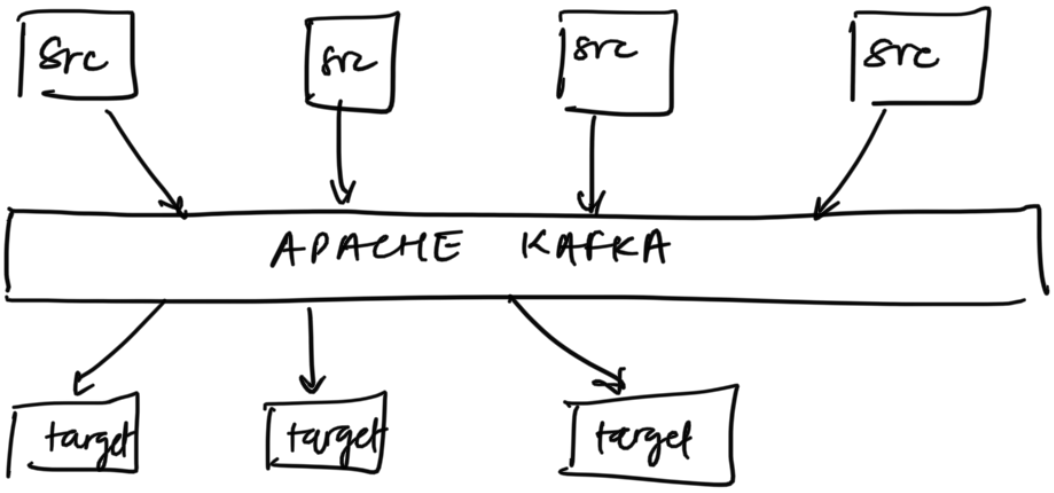
- Previously -



- Very complicated!

- If you 4 src and 6 target you need 6x4 = 24 integrations
- Each integration has a lot of difficulties
  - Protocol, Data format, Data schema & evolution.
- Each source system will have an increased load from the connections.

* Apache Kafka -
- allows to decouple data stream and the system.
- software platform based on distributed streaming process. It is a publish-subscribe messaging system which lets exchanging of data between applications, servers and processes as well.



- data from src will end up in kafka
- target system will source their data from kafka

* Why Apache kafka?
- Created by Linkedin, now Open Source Project mainly maintained by confluent
- Distributed, resilient architecture, fault tolerant
- Horizontal scalability
  - can scale to 100's of brokers
  - can scale to millions of messages per second
- High performance (latency less than 10ms) - real time
- Used by 2000+ firms, 35% of fortune 500 - netflix, walmart, uber etc.

* Use cases:
- Messaging System
- Activity tracking
- Gather metrics from many different locations

- Application logs gathering.
- Stream processing (Kafka Streams API or Spark)
- Decoupling of system dependencies.
- Integration with Spark, Flink, Storm, Hadoop and many other Big Data Technologies.

Apache Kafka's - core API's

1) Producer API - allows application to publish streams of records to one or more topics.

2) Consumer API - allows an application to subscribe one or more topics and process the stream of records produced to them.

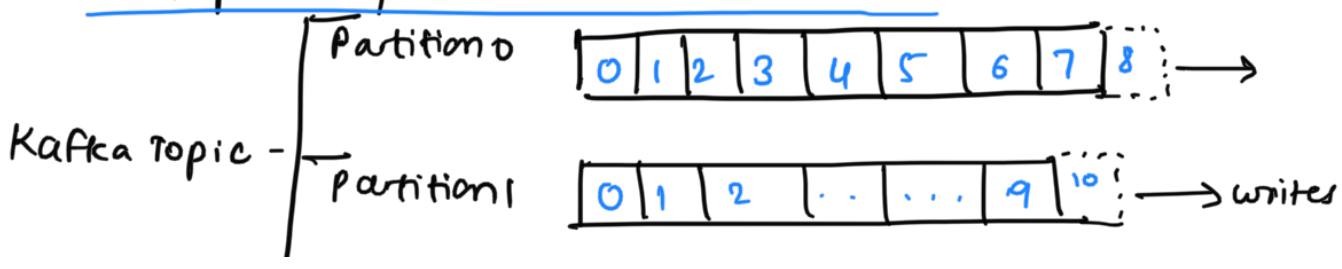
3) Streams API - allows an application to effectively transform input streams to output streams.
   - permits an application to act as stream processor which consumes an input stream from one or more topics and produce an output stream to one or more output topics.

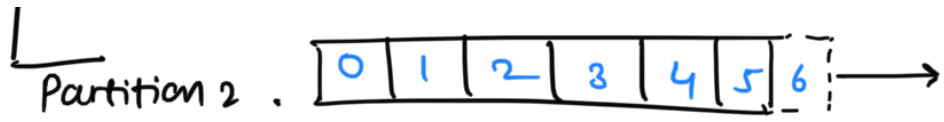
4) Connector API - This API executes the reusable producer and Consumer API with existing data systems or applications.

Topics, Partition, Offset.

Topics: a particular stream of data
- Similar to table in a database (without all constraints)
- you can have as many topics as you want.
- A topic is identified by its name

- Topics are split into partitions
  - Each partition is ordered
  - Each message within a partition gets an incremential id, called offset.
  - offset only have a meaning for a specific partition
    eg: offset 3 in partition 0 doesn't represent the same data as offset 3 in partition 1.
  - Order is guaranted only within a partition (not across partitions)
  - Data is kept only for a limited time (1 week)





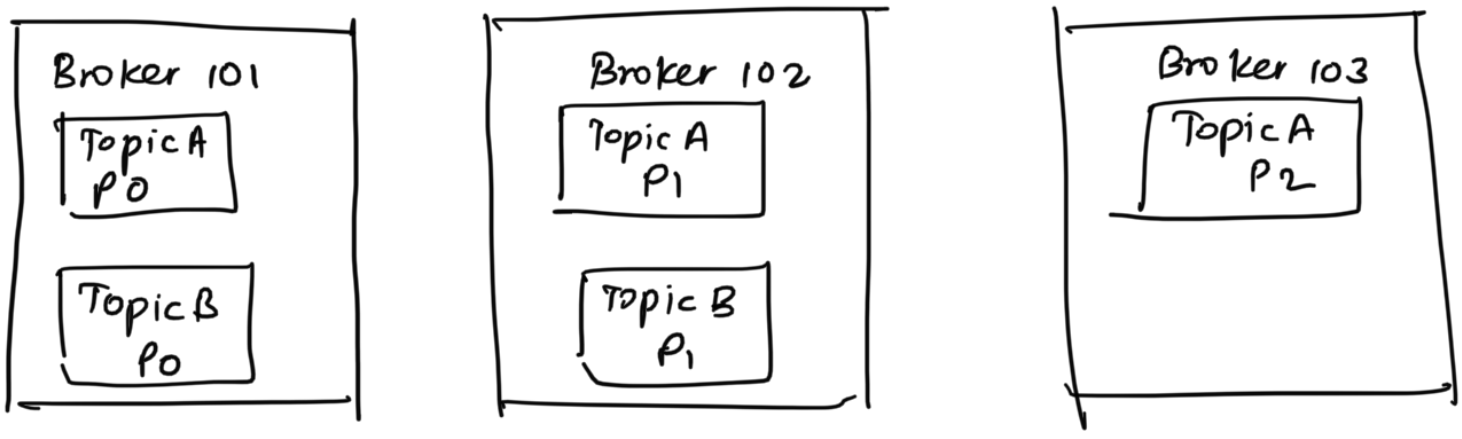
- Once data is written to a partition it cant be changed.
- Data is assigned randomly to a partition unless a key is provided

Brokers

- A Kafka cluster is comprised of multiple brokers (servers).
- Each broker is identified with its ID (integer)
- Each broker contains certain topic partitions
* After connecting to any broker (called a bootstrap broker) you will be connected to entire cluster.

Brokers and Topics

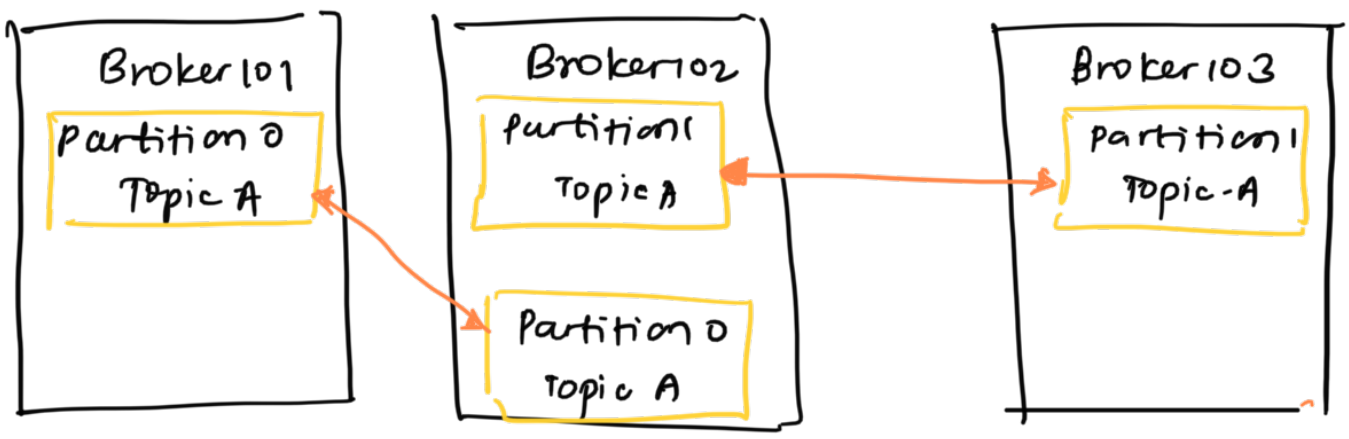
Topic A – 3 partitions (P0, P1, P2)
Topic B - 2 partitions . (P0, P1)



Topic Replication Factor

- >1 (usually b/w 2 and 3)
- This way if one broker is down, another broker can serve the data.

eg. Topic A - 2 partition, replication factor 2



Concept of leader for a partition

- At any time only ONE broker can be a leader for a given partition.
- Only that leader can receive and serve data for a partition.
- The other brokers will synchronize the data
- Therefore each partition has one leader and multiple in sync replicas. (ISR)

Producer.

- writes data to topics (which is made of partitions)
- producer automatically know to which broker and partition to write to.
- In case of broker failures, producers will automatically recover
- producer will load balance in absence of keys. It will divide data between different partitions.

Producer can receive acknowledgement of data writes.

acks = 0 : producer wont wait for ack (possible data loss).

acks = 1 : will wait for leader aek. (limited data loss).

acks = all : leader + replicas ack (no data loss)

Message keys:

- producer can choose to send a key with the message (string, number)
- If key = null, data is sent round robin. (101 → 102 → 103 ...)
- If key is sent, then all messages for that key will go to the same partition always.
- A key is basically sent if you need message ordering for a specific field. (eg. truck_id)

Consumer.

- read data from a topic (identified by name)
- consumers know which broker to read from.
- In case of broker failures, consumers know how to recover.
- Data is read in order within each partitions.

consumer groups:

- consumer reads data in consumer groups.
- each consumer within a group reads from exclusive partitions.
- If you have more consumers than partitions, some will be inactive.
- Consumers know where to read from, they will automatically use a group co-ordinator and a consumer co-ordinator to assign consumers to a partition.

Consumer offsets:

- Kafka stores offsets at which a consumer group has been reading.
- The offsets commited live in a kafka topic named __consumer_offsets.
- when a consumer in a group has processed data received from kafka it should be committing the offsets.
- If a consumer dies it will be able to read back from where it left off, thanks to committed consumer offsets.

Consumers choose when to commit offsets.

There are 3 delivery semantics -

At most once - offsets are committed as soon as the message is received.
- If processing goes wrong, the message will be lost (it won't be read again)

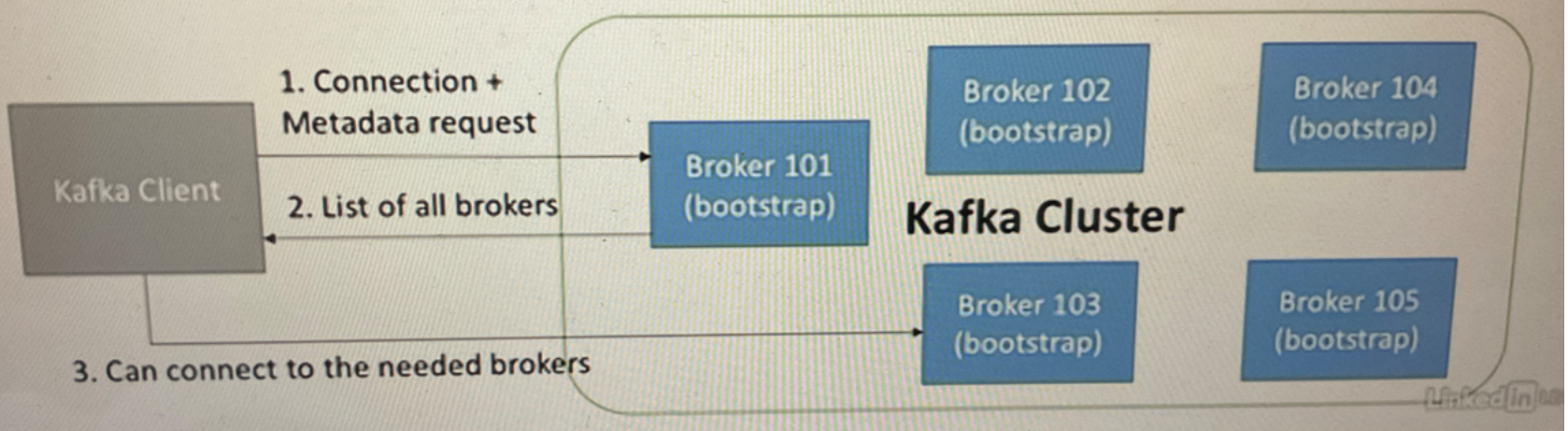
Atleast once - offsets are comitted after the message is processed
- if processing goes wrong, the message will be read again.
- This can result in duplicate processing of msgs.

So make sure your processing is idempotent
(ie - processing again the messages wont impact your systems).

Exactly once: Can be achieved for kafka => kafka workflows using kafka streams API.
For kafka => external system workflows, use an idempotent consumer.

Kafka Broker Discovery
- Each broker is also called a bootstrap broker
- This means you only need to connect to one broker. and you will be connected to entire cluster.
- Each broker knows about all brokers, topics and partitions (metadata).

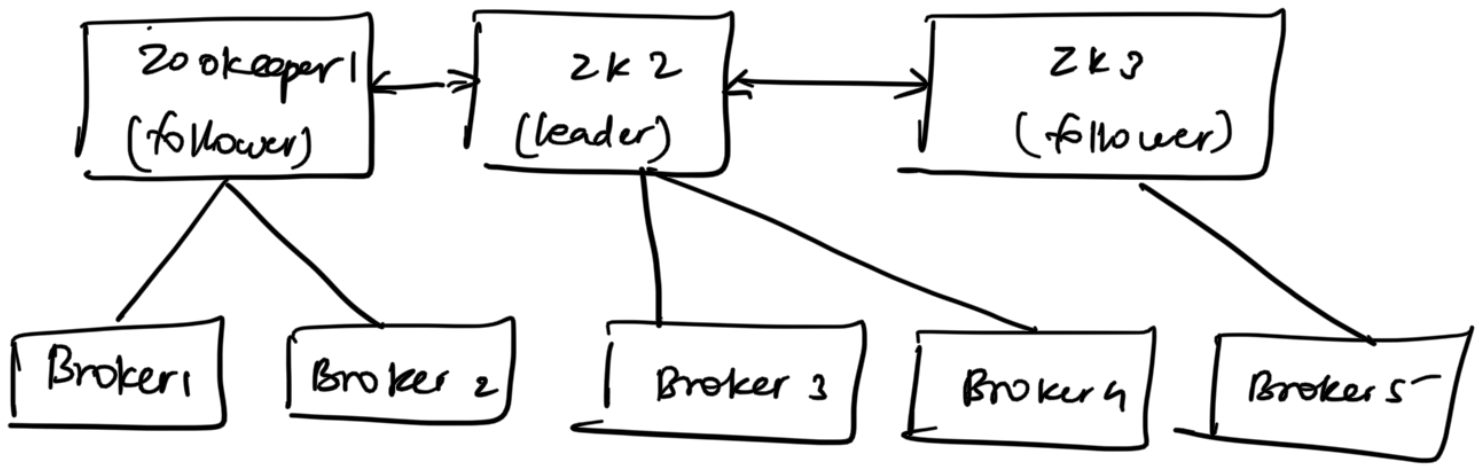
- Every Kafka broker is also called a "bootstrap server"
- That means that **you only need to connect to one broker**, and you will be connected to the entire cluster.
- Each broker knows about all brokers, topics and partitions (metadata)



to get connected to the entire thing.

MacBook Air

Zookeeper:
- manages brokers (keeps a list of them)
- helps in performing leader elections for partitions.
- Sends notifications to kafka in case of changes (eg. new topic, broker dies, broker comes up etc)
- kafka cant work without zookeeper.
- zookeeper by design operates on an odd number of servers (3,5,7).
- zookeeper has a leader (handle writes) the rest of the servers are followers.
- zookeeper does not store consumer offset with kafka > v0.10)



Kafka Guarantees:

Kafka Guarantees

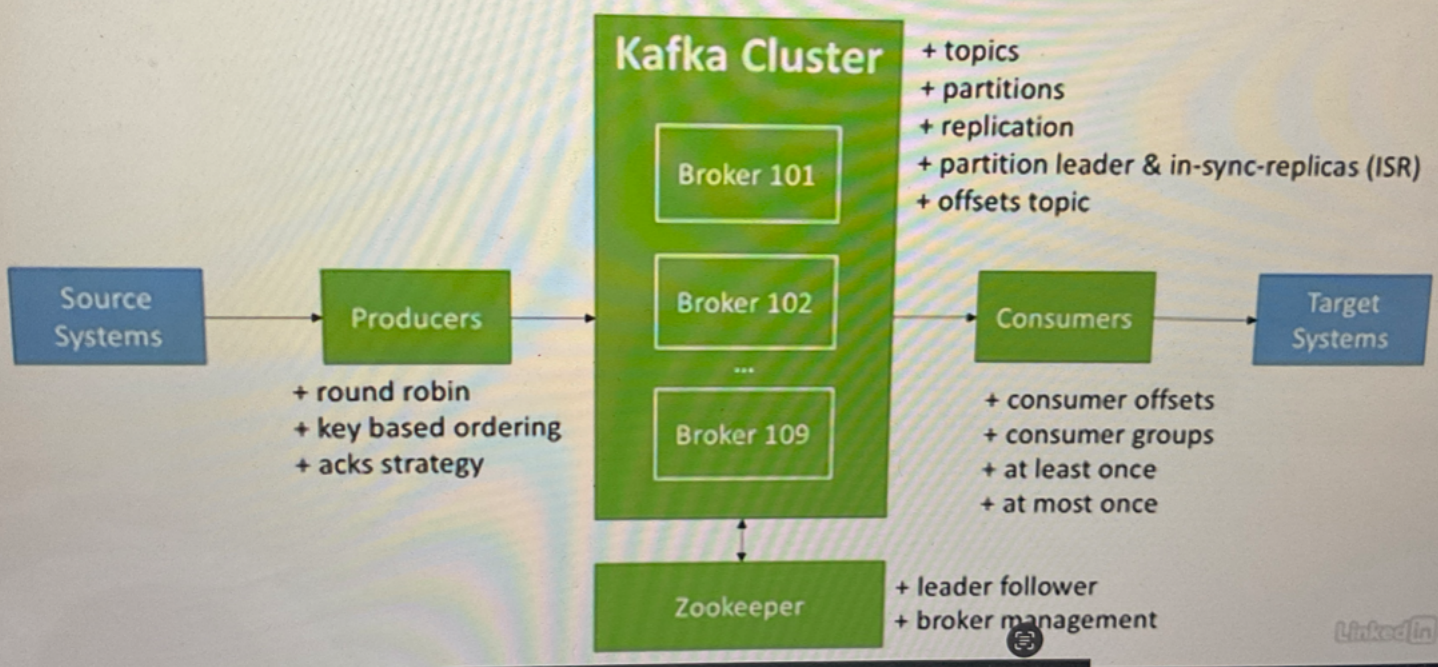
- Messages are appended to a topic-partition in the order they are sent
- Consumers read messages in the order stored in a topic-partition
- With a replication factor of N, producers and consumers can tolerate up to N-1 brokers being down
- This is why a replication factor of 3 is a good idea:
  - Allows for one broker to be taken down for maintenance
  - Allows for another broker to be taken down unexpectedly
- As long as the number of partitions remains constant for a topic (no new partitions), the same key will always go to the same partition

you'll remember them, I promise,

Theory Round up:

418 active

Theory Roundup
We've looked at all the Kafka concepts



with one broker and one Zookeeper on our machine

1:53 / 1:58

1.75x