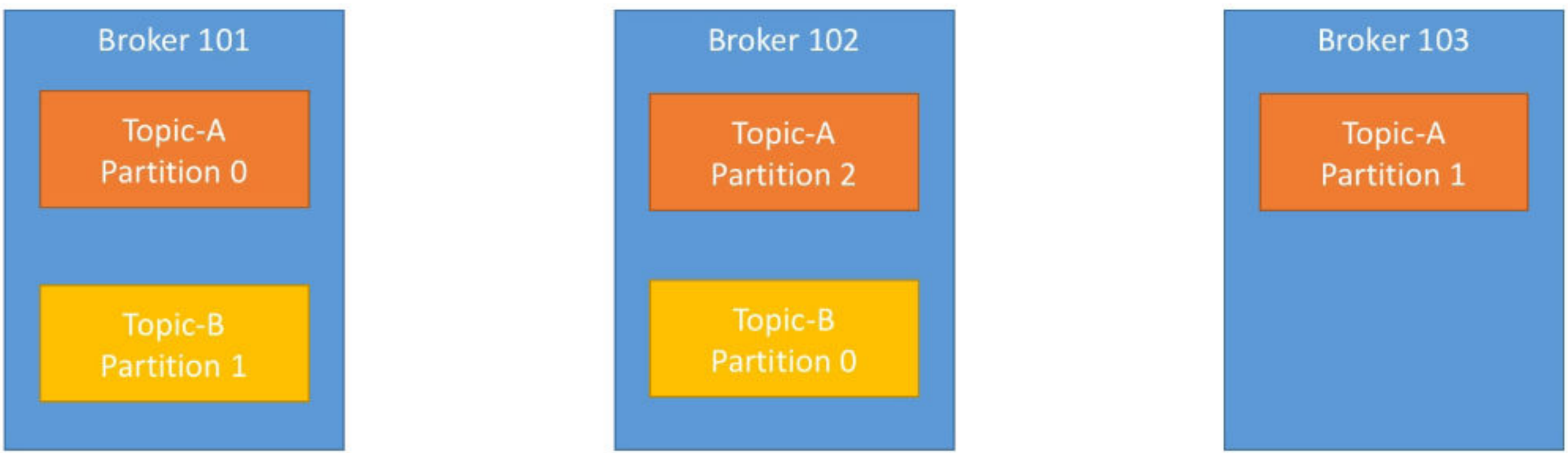
Brokers

- A Kafka cluster is composed 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 the entire cluster
- A good number to get started is 3 brokers, but some big clusters have over 100 brokers
- In these examples we choose to number brokers starting at 100 (arbitrary)

Brokers and topics

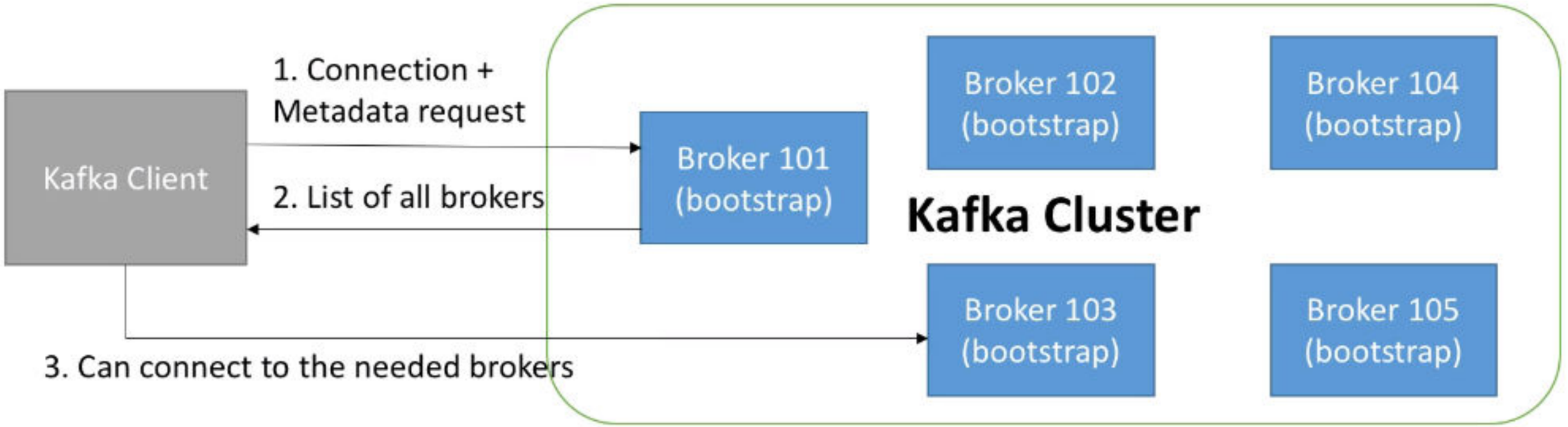
- Example of **Topic-A** with **3 partitions**
- Example of **Topic-B** with **2 partitions**



- Note: Data is distributed and Broker 103 doesn't have any **Topic B** data

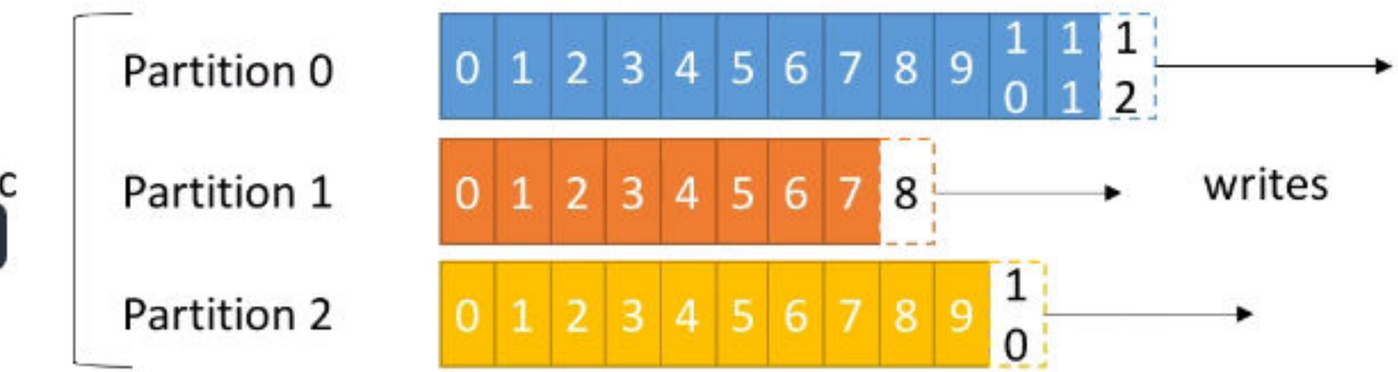
Kafka Broker Discovery

- 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)



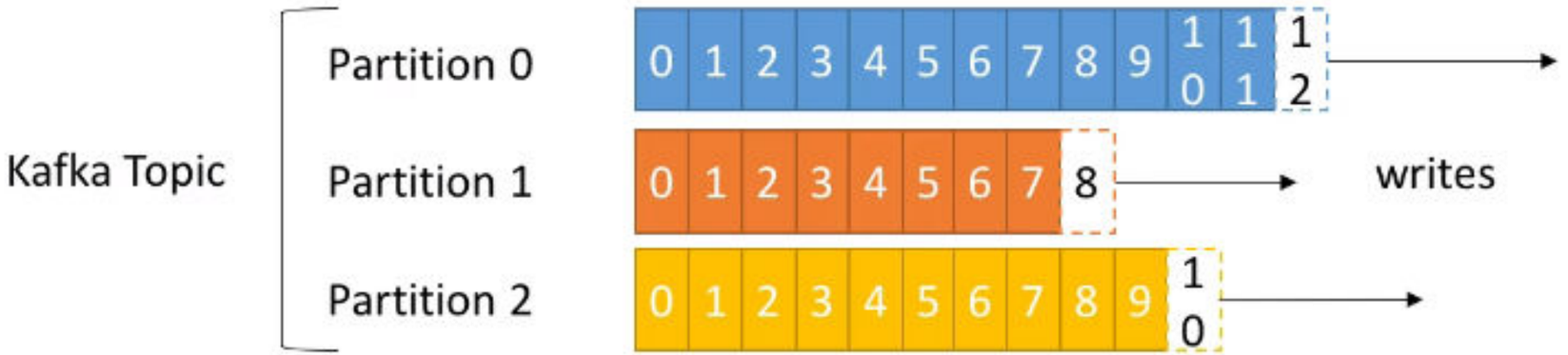
Topics, partitions and offsets

- Topics: a particular stream of data
  - Similar to a table in a database (without all the constraints)
  - You can have as many topics as you want
  - A topic is identified by its name
- Topics are split in partitions
  - Each partition is ordered
  - Each message within a partition gets an incremental id, called offset



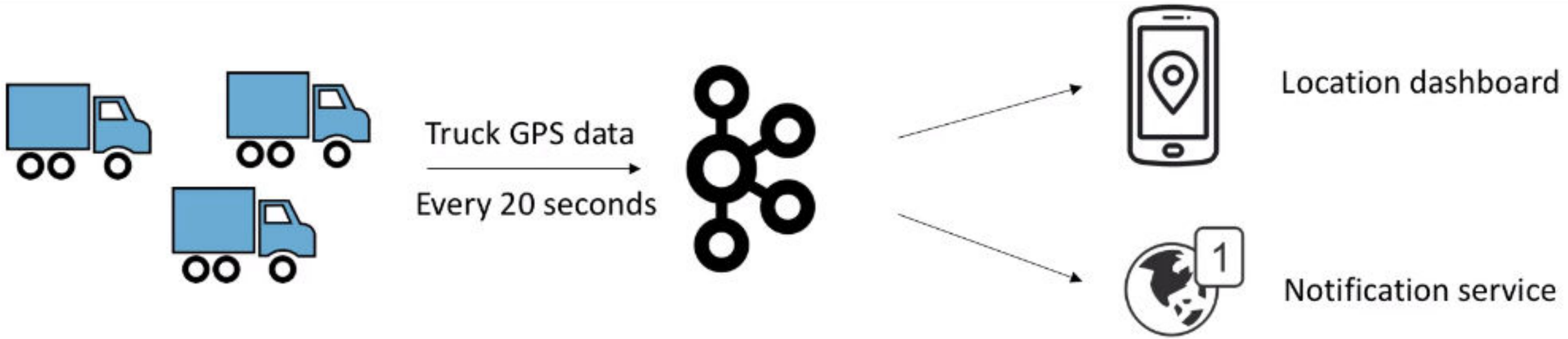
e have written a note here.

Topics, partitions and offsets



- Offset only have a meaning for a specific partition.
  - E.g. offset 3 in partition 0 doesn't represent the same data as offset 3 in partition 1
- Order is guaranteed only within a partition (not across partitions)
- Data is kept only for a limited time (default is one week)
- Once the data is written to a partition, it can't be changed (immutability)
- Data is assigned randomly to a partition unless a key is provided (more on this later)

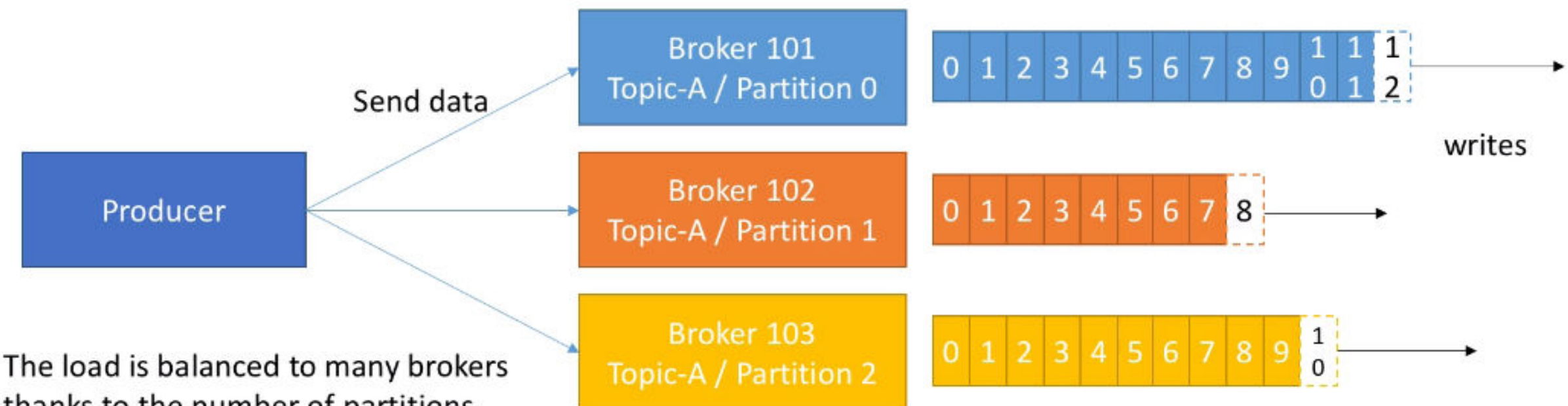
Topic example: truck_gps



- Say you have a fleet of trucks, each truck reports its GPS position to Kafka.
- You can have a topic trucks_gps that contains the position of all trucks.
- Each truck will send a message to Kafka every 20 seconds, each message will contain the truck ID and the truck position (latitude and longitude)
- We choose to create that topic with 10 partitions (arbitrary number)

Producers

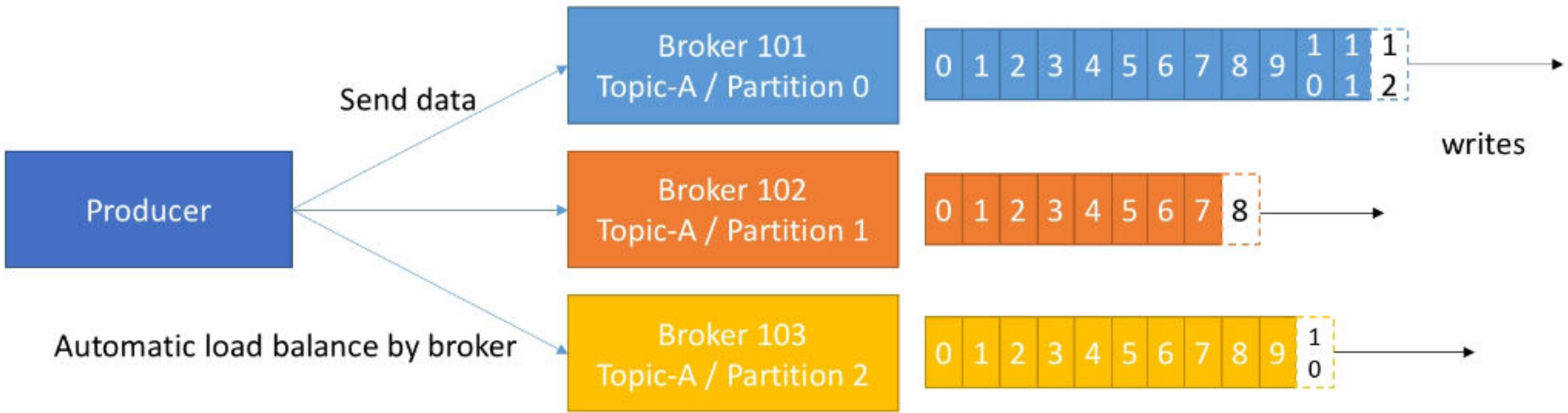
- Producers write data to topics (which is made of partitions)
- Producers automatically know to which broker and partition to write to
- In case of Broker failures, Producers will automatically recover



The load is balanced to many brokers thanks to the number of partitions

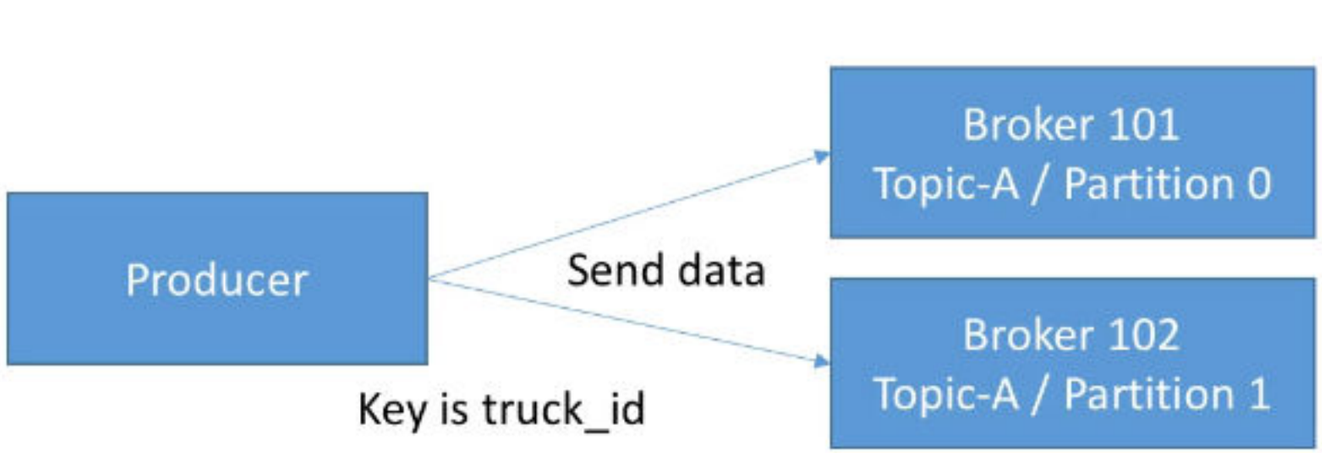
Producers

- Producers can choose to receive acknowledgment of data writes:
  - acks=0: Producer won't wait for acknowledgment (possible data loss)
  - acks=1: Producer will wait for leader acknowledgment (limited data loss)
  - acks=all: Leader + replicas acknowledgment (no data loss)



Producers: Message keys

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



Example:

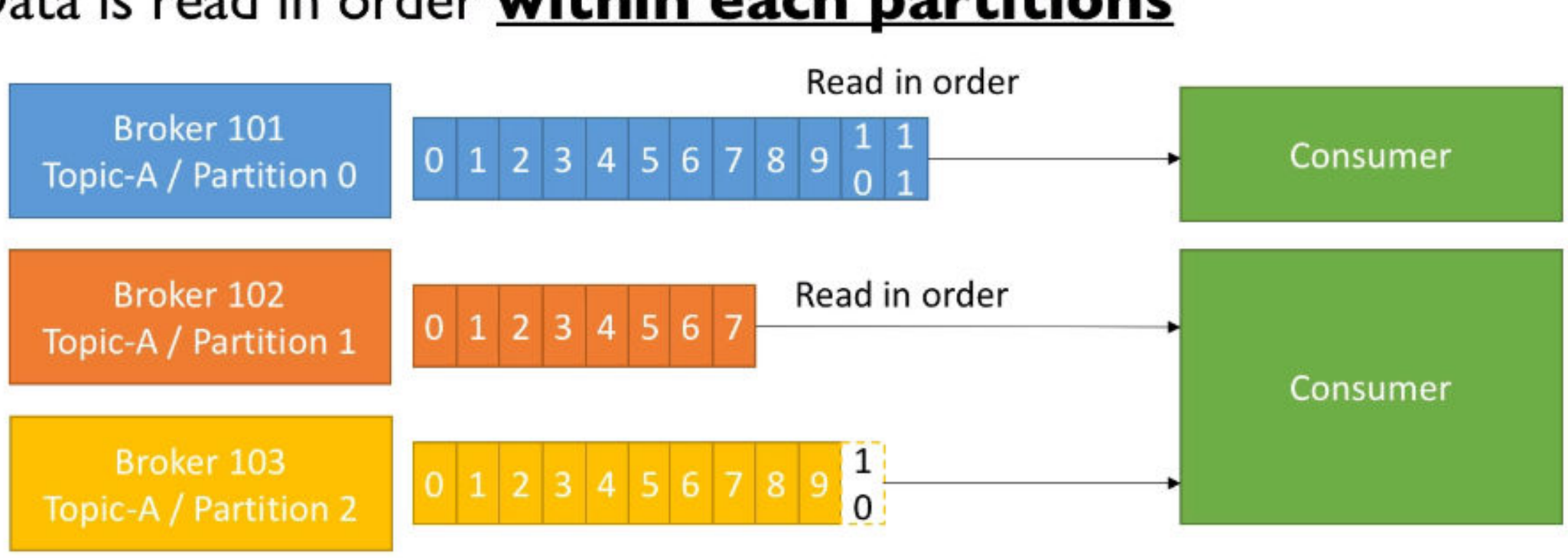
truck_id_123 data will always be in partition 0
truck_id_234 data will always be in partition 0

truck_id_345 data will always be in partition 1
truck_id_456 data will always be in partition 1

(Advanced: we get this guarantee thanks to key hashing, which depends on the number of partitions)

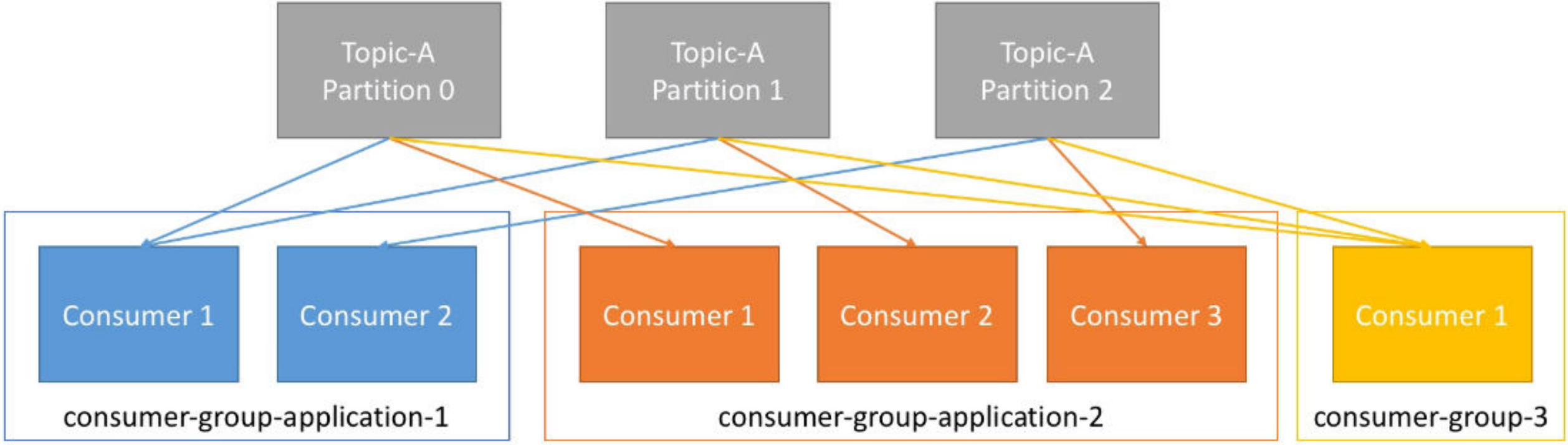
Consumers

- Consumers 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

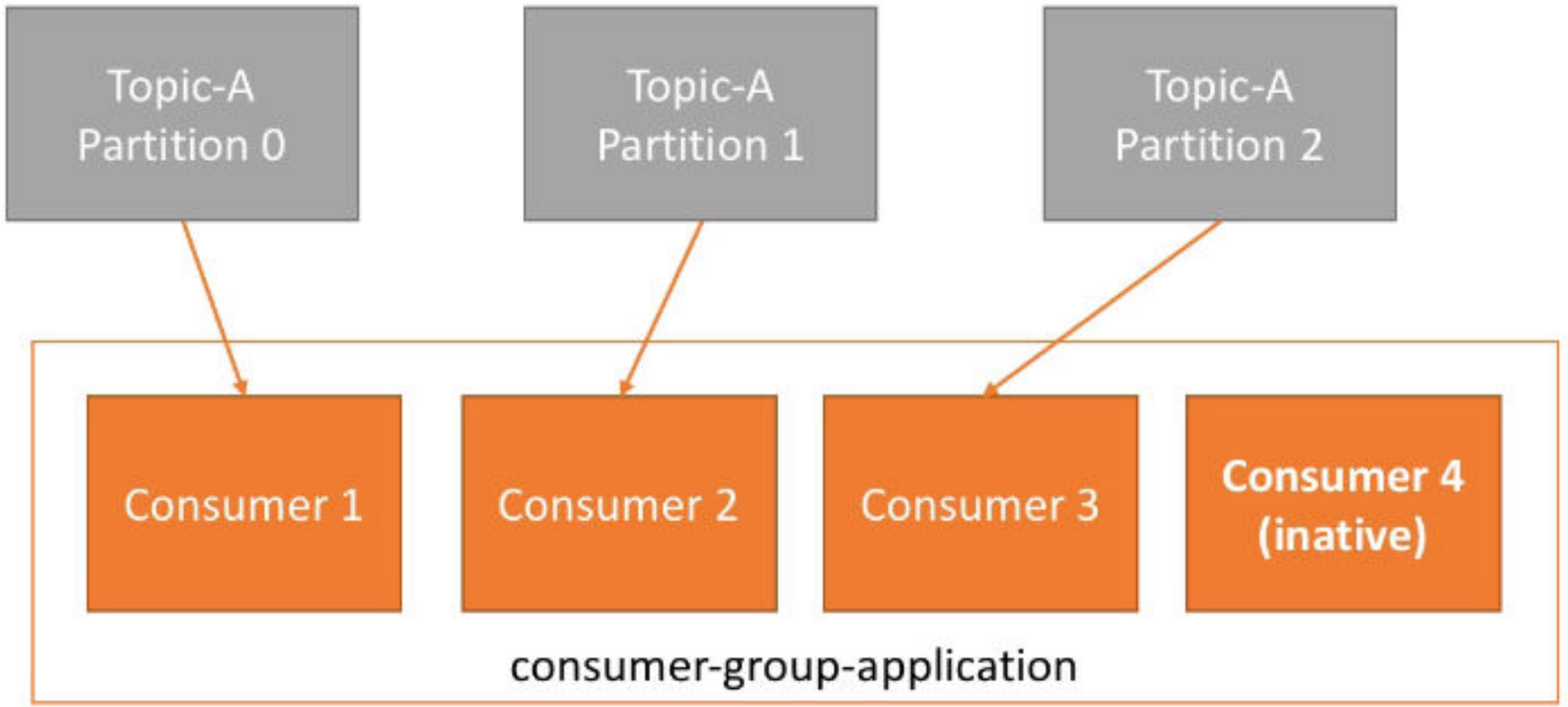
- Consumers read data in consumer groups
- Each consumer within a group reads from exclusive partitions
- If you have more consumers than partitions, some consumers will be inactive



- *Note: Consumers will automatically use a GroupCoordinator and a ConsumerCoordinator to assign a consumers to a partition.*

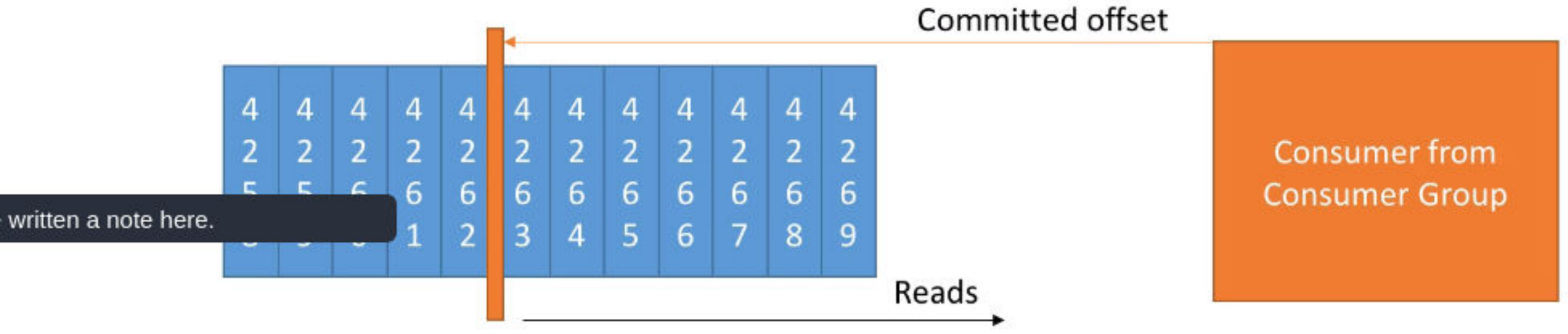
Consumer Groups
What if too many consumers?

- If you have more consumers than partitions, some consumers will be inactive



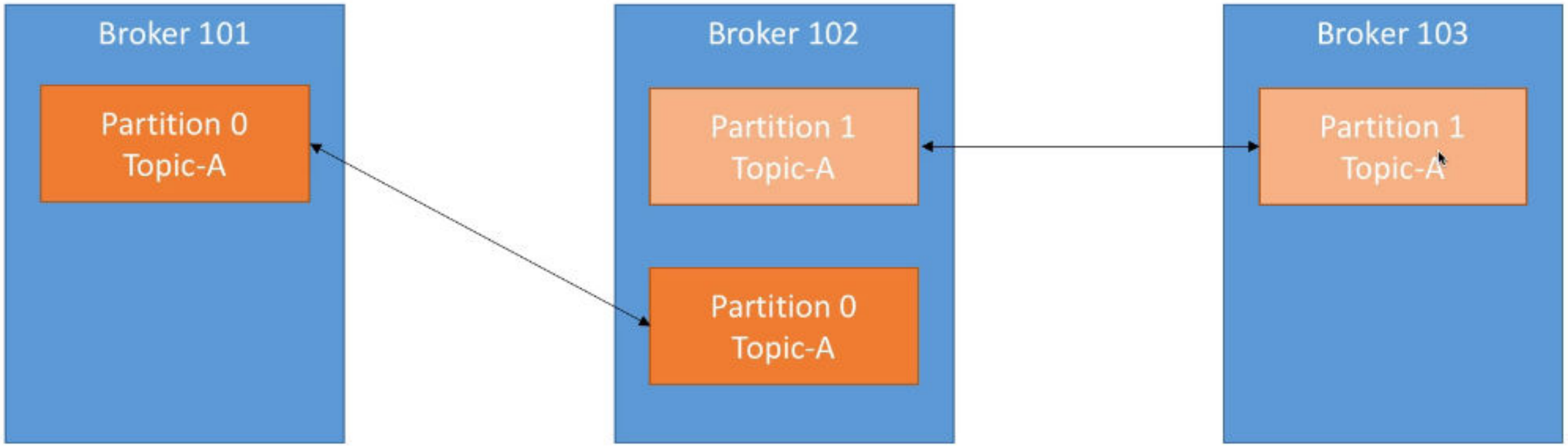
Consumer Offsets

- Kafka stores the offsets at which a consumer group has been reading
- The offsets committed 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 the committed consumer offsets!



Topic replication factor

- Topics should have a replication factor > 1 (usually between 2 and 3)
- This way if a broker is down, another broker can serve the data
- Example: Topic-A with 2 partitions and replication factor of 2

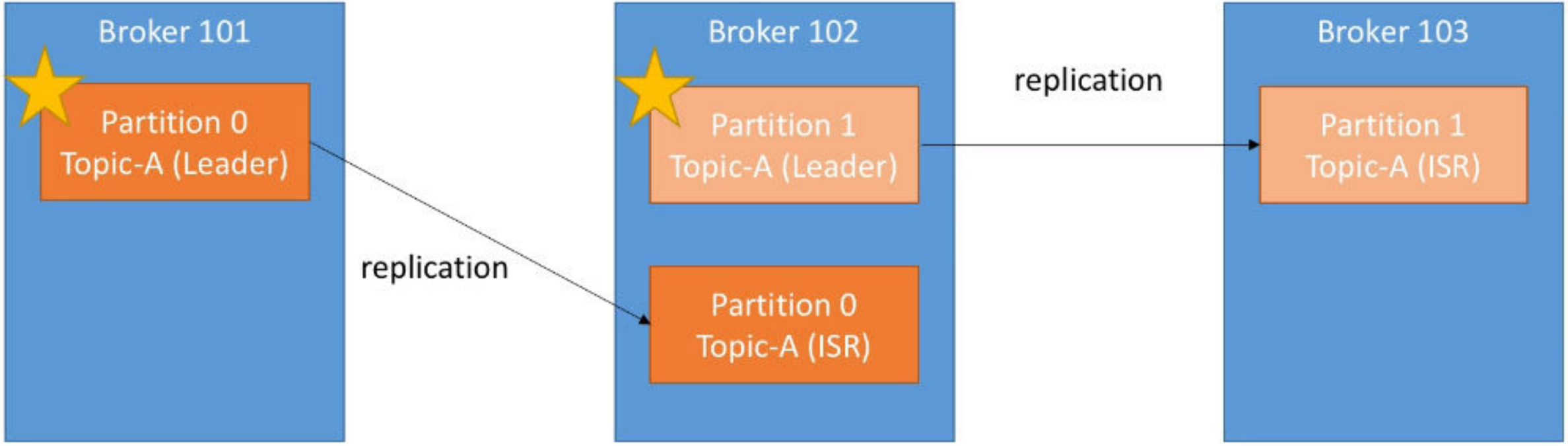


Topic replication factor

- Example: we lost Broker 102
- Result: Broker 101 and 103 can still serve the data

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 ISR (in-sync replica)



Delivery semantics for consumers

- 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 the processing goes wrong, the message will be lost (it won't be read again).
- At least once (usually preferred):
  - offsets are committed after the message is processed.
  - If the processing goes wrong, the message will be read again.
  - This can result in duplicate processing of messages. Make sure your processing is idempotent (i.e. processing again the messages won't 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.

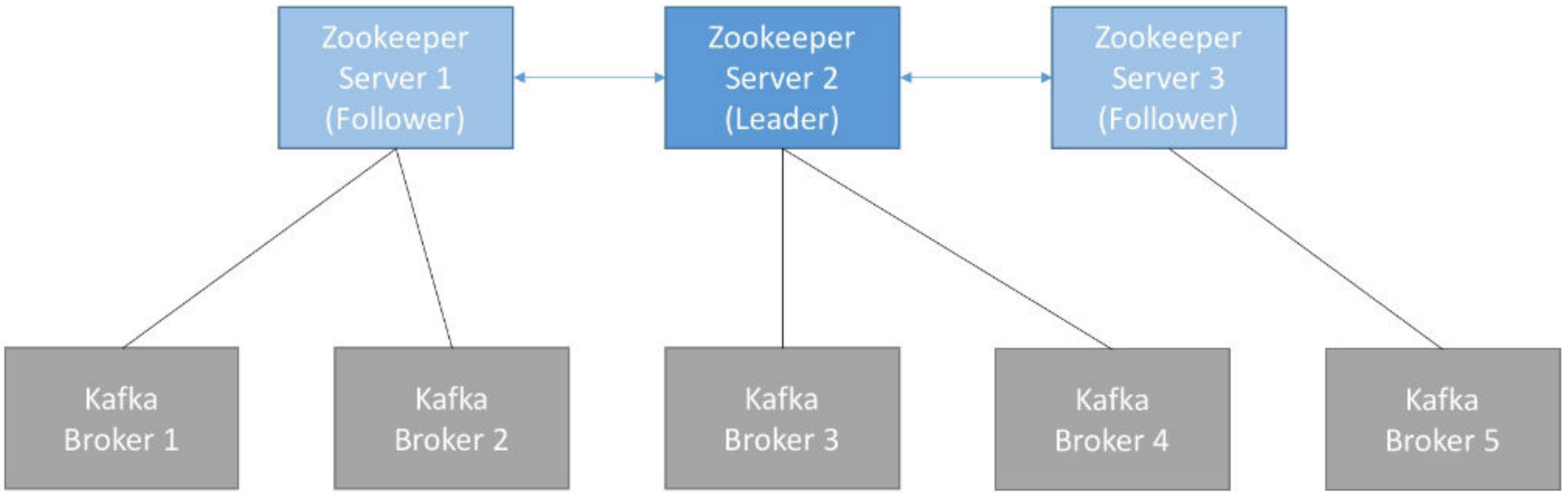
Kafka Guarantees

- 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

Zookeeper

- Zookeeper manages brokers (keeps a list of them)
- Zookeeper helps in performing leader election for partitions
- Zookeeper sends notifications to Kafka in case of changes (e.g. new topic, broker dies, broker comes up, delete topics, etc.…)
- **Kafka can't work without Zookeeper**
- Zookeeper by design operates with an odd number of servers (3, 5, 7)
- Zookeeper has a leader (handle writes) the rest of the servers are followers (handle reads)
- (Zookeeper does NOT store consumer offsets with Kafka > v0.10)

Zookeeper


Zookeeper Server 1 (Follower)
Zookeeper Server 2 (Leader)
Zookeeper Server 3 (Follower)
Kafka Broker 1
Kafka Broker 2
Kafka Broker 3
Kafka Broker 4
Kafka Broker 5