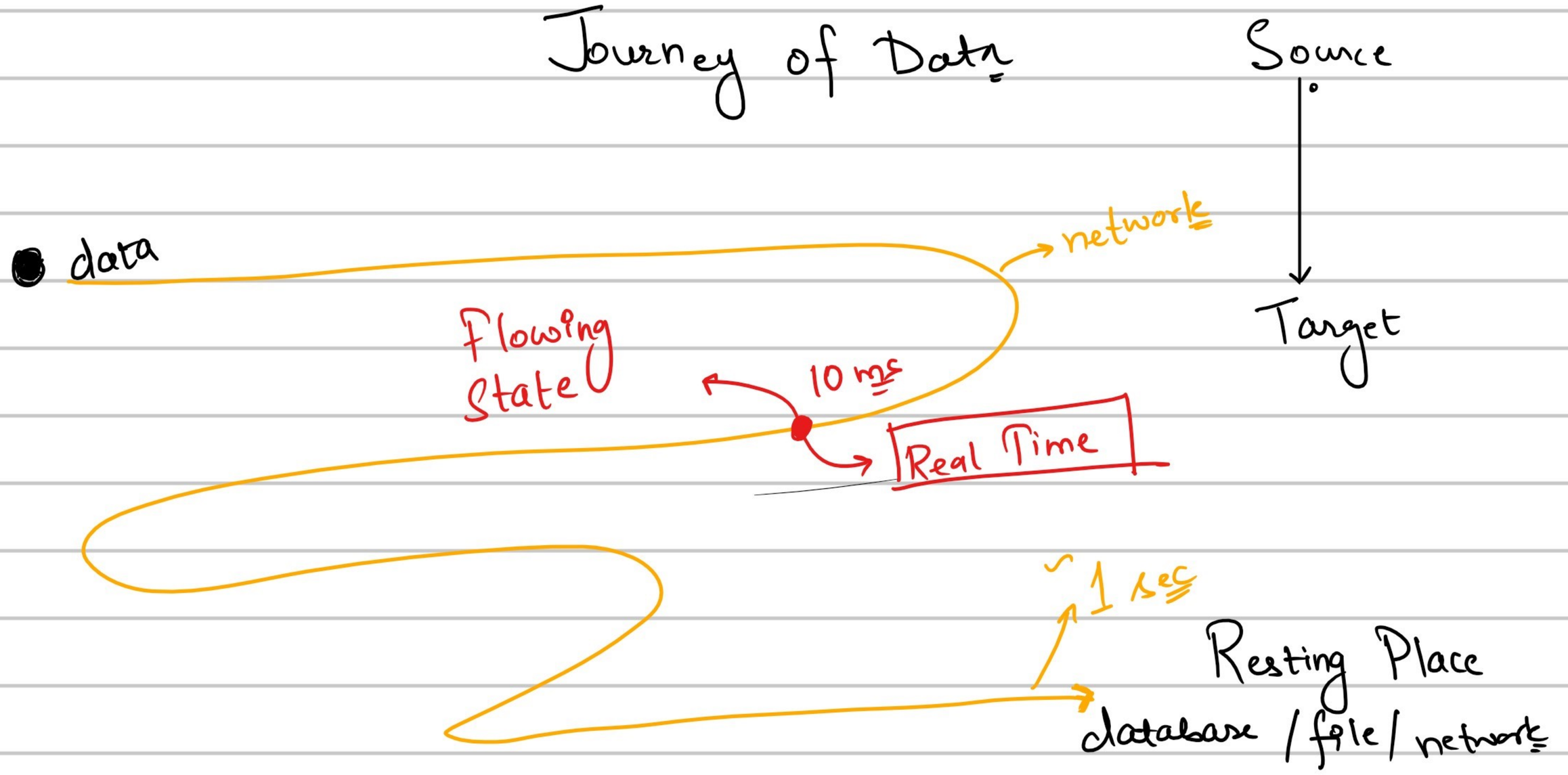
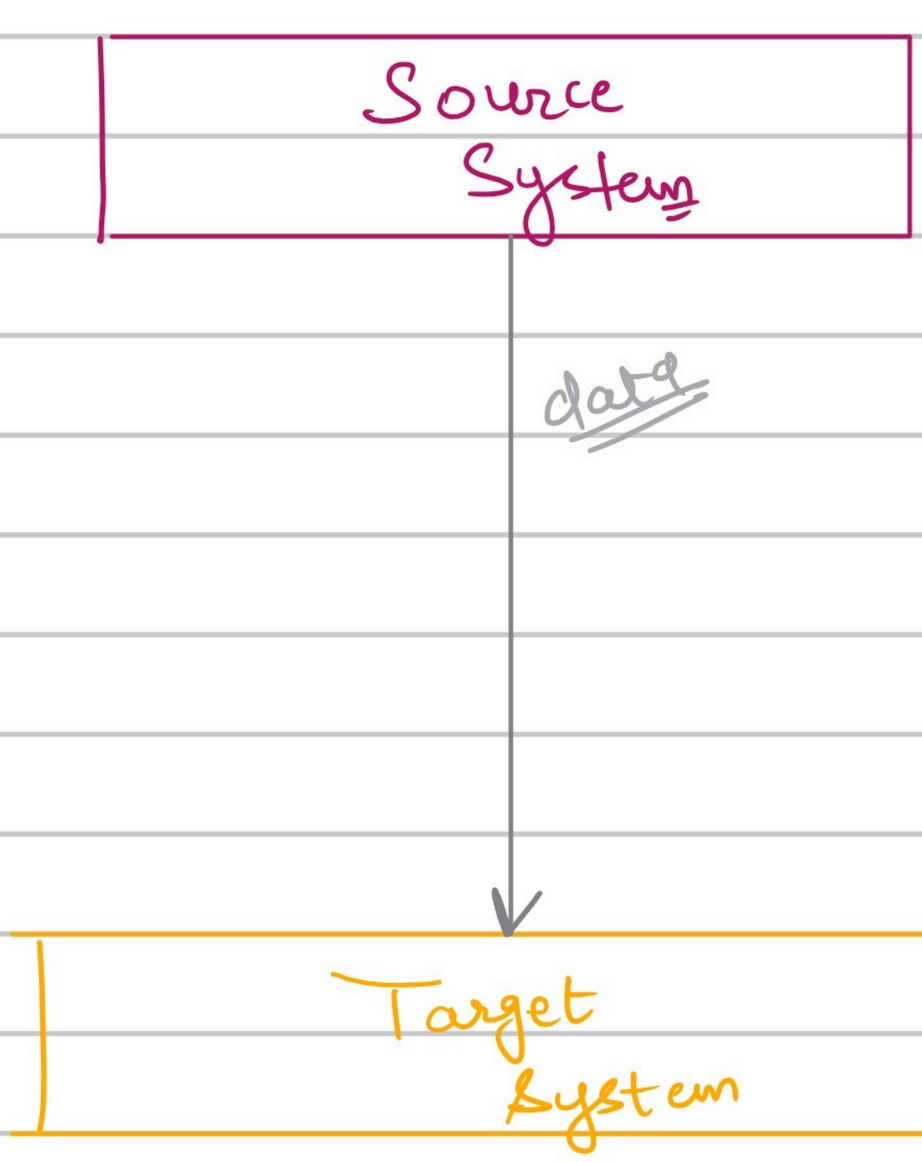
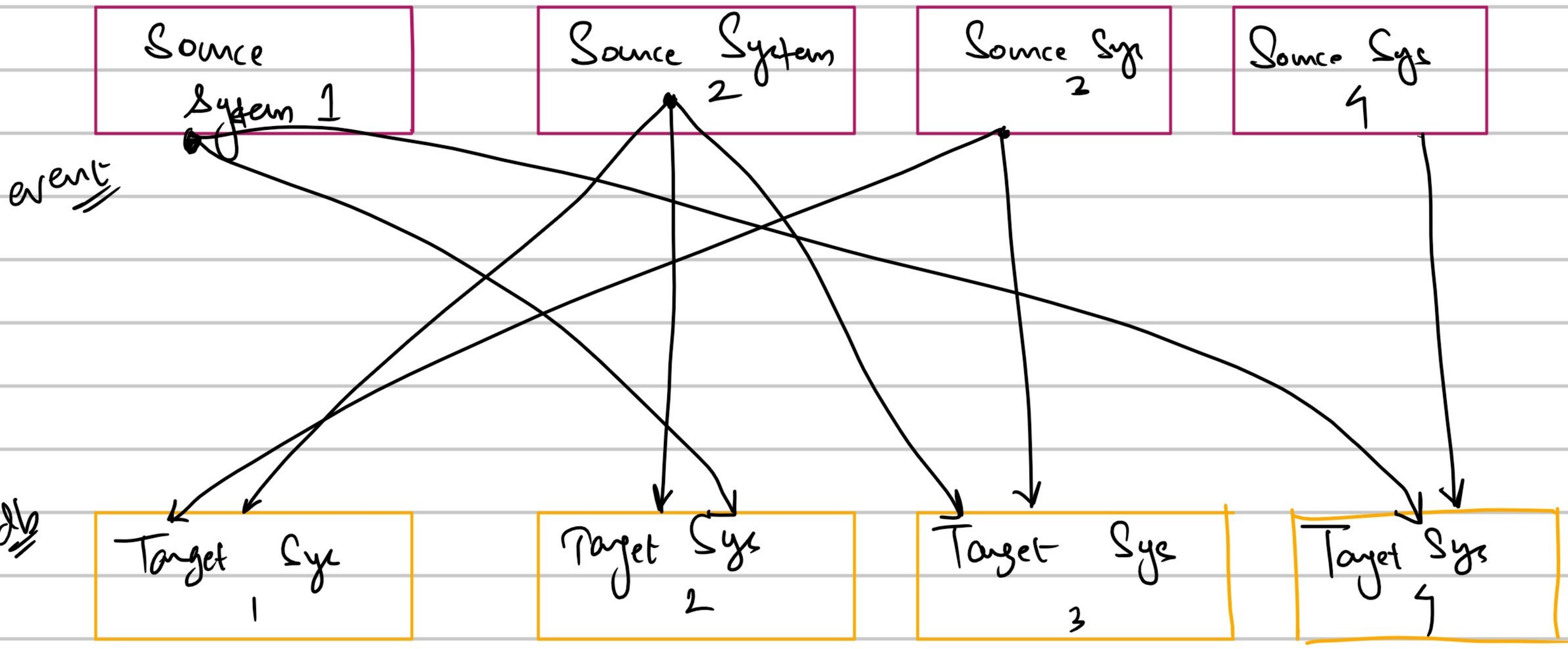
Kafka → messaging queue

Journey of Data





Simple at first !!



After a while it becomes complicated !!

4 source system x 6 target systems

4 x 6 (24) integrations

Challenges we will face:

* Protocol → TCP/HTTP/JDBC/REST [How data is transported]

* Data Format → Data is passed in which format [Binary, CSV, Avro...]

* Schema → How the data is shaped

* One more source is introduced *
↓
It will increase the network load
↳

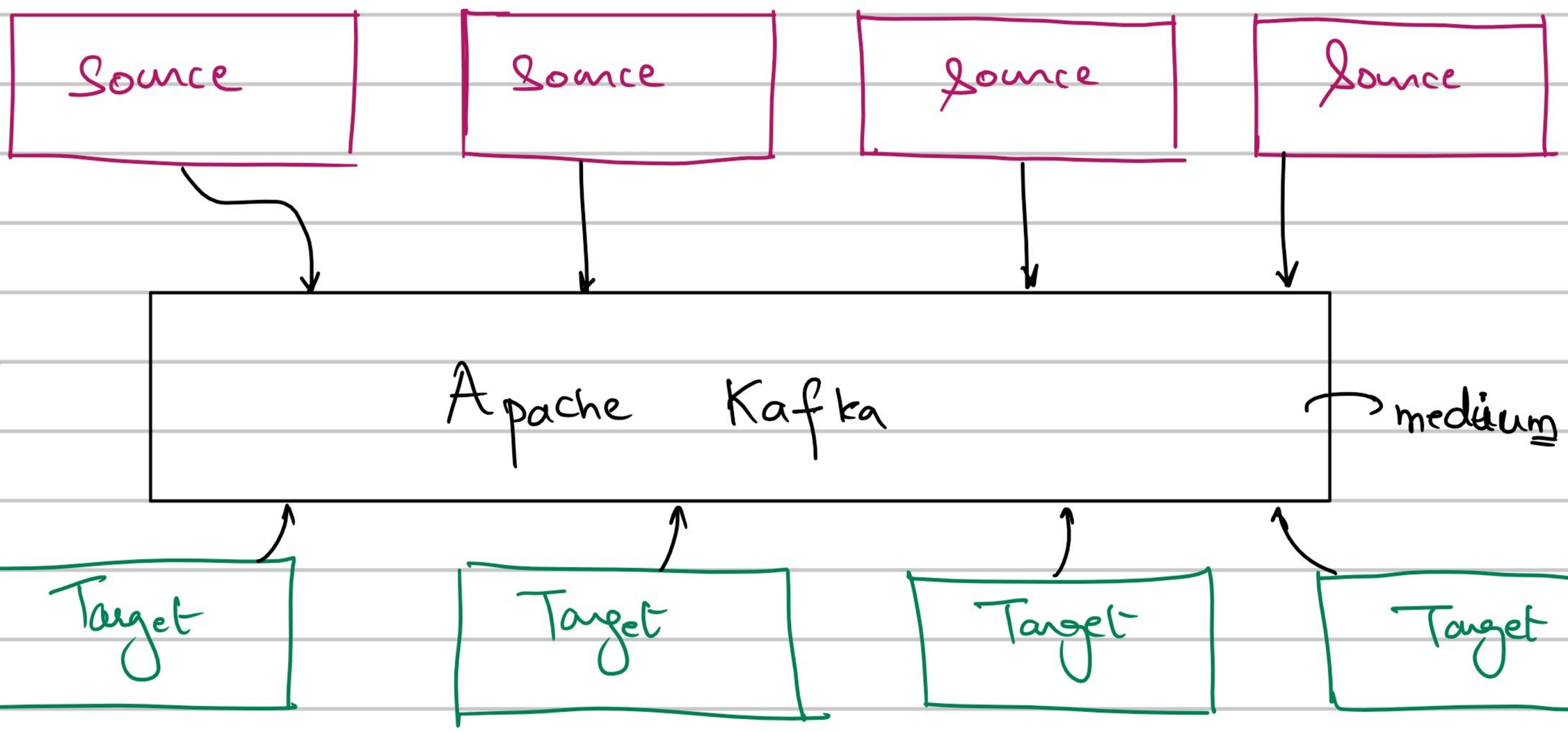
Source & Target system have a very tight coupling

Apache Kafka → Decoupling of data streams

pub-sub model

Publisher ↓ Subscriber

pattern where a publisher sends a piece of data → not directly to the reciever ; infact it will send it through a medium.



Why Kafka?

↳ Created at LinkedIn, → open source project

↓

→ Distributed, Resilient, fault Tolerant Confluent

* Horizontal Scaling → 100's of brokers
→ Can scale to millions of messages per second.

* High Performance → latency of less than 10 ms → real time

Apache Kafka – Use Cases

* Messaging System
* Activity Tracking
* Stream Processing

- Gather metrics from diff locations
- Gather application logs
- Integrated with big data ~~system~~ modules

Kafka is just a transportation tool !!

3 big differences which set Kafka aside.

① Distributed System → rather than running dozens of individual messaging brokers; hand wired to different apps.

We have a central platform that can scale!!

② Kafka is a true storage system → it is built to store the data as long as you want
(default → 1 week)

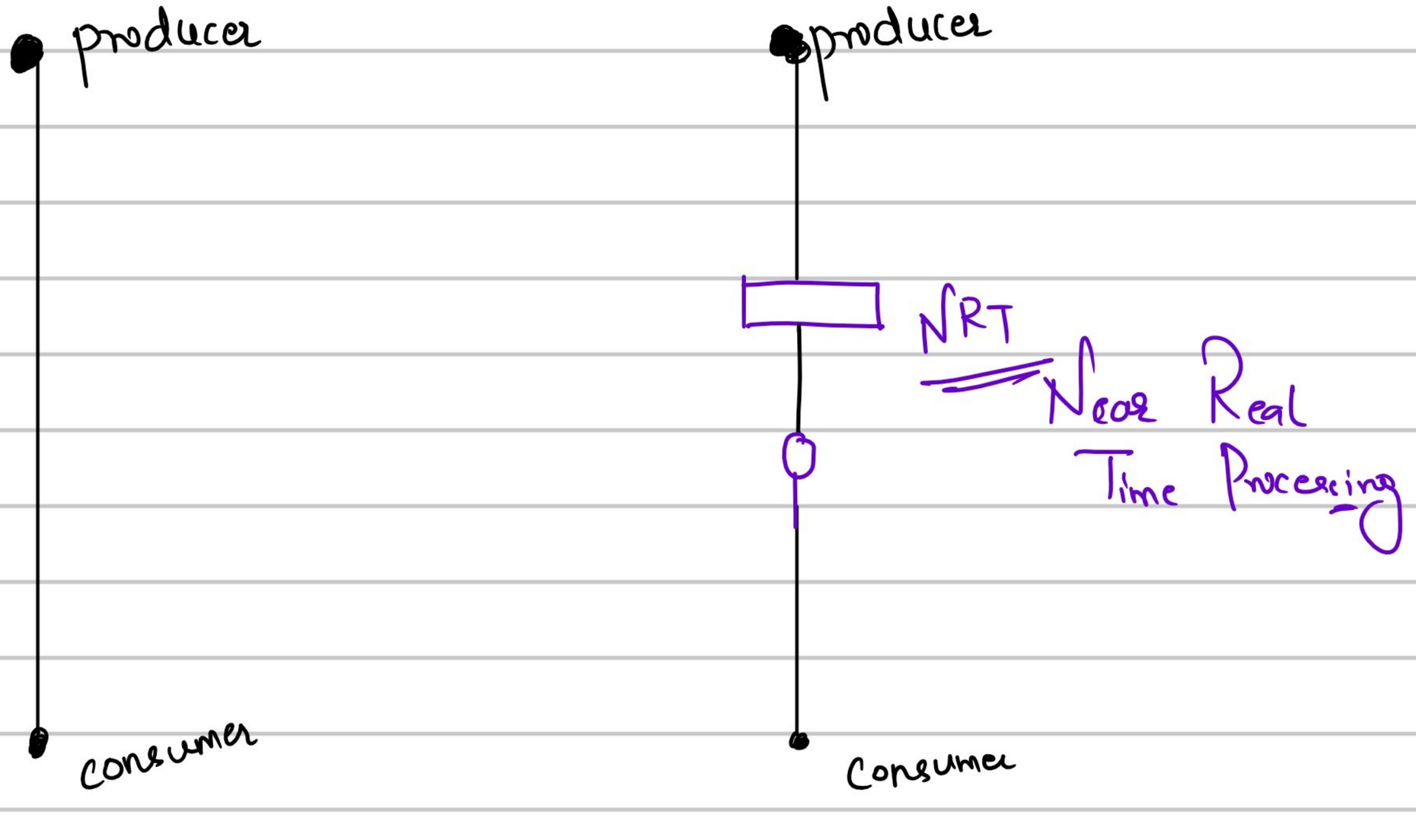
Real Delivery Guarantees → [replicated | persisted]

③ Kafka supports Stream Processing

→ Other Messaging System → just handout the messages.

With Stream Processing in Kafka

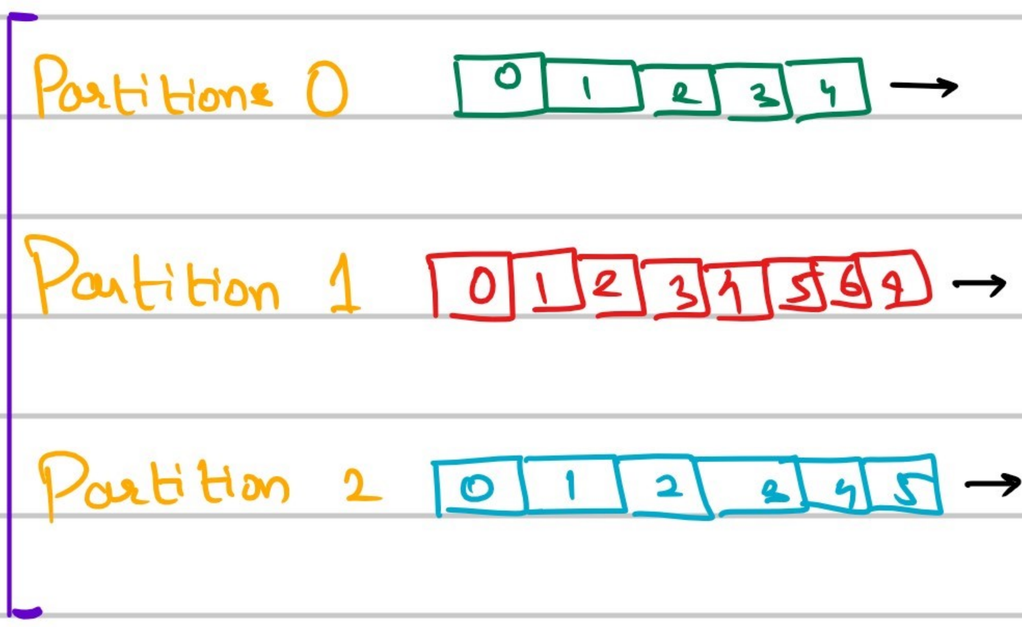
↳ allows us to compute derived stream & generate more meaningful/rich data



Topics, Partitions & Offsets

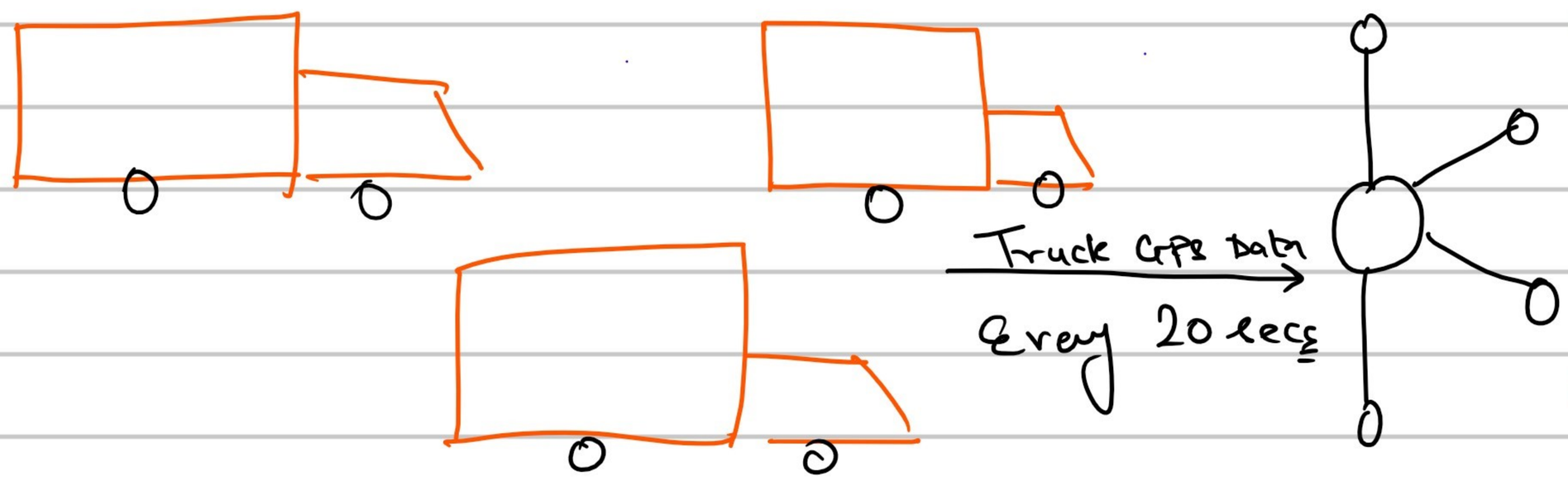
* Topics → a particular stream of data
  - Similar to a table in database (without all constraints)
  → You can have as many as topics you want.

→ A topic is identified by its name



* Topics are split into Partitions.
  * Each partition is ordered
  * Each message in a partition gets an incremental id, called as offset

Topic → truck_gps

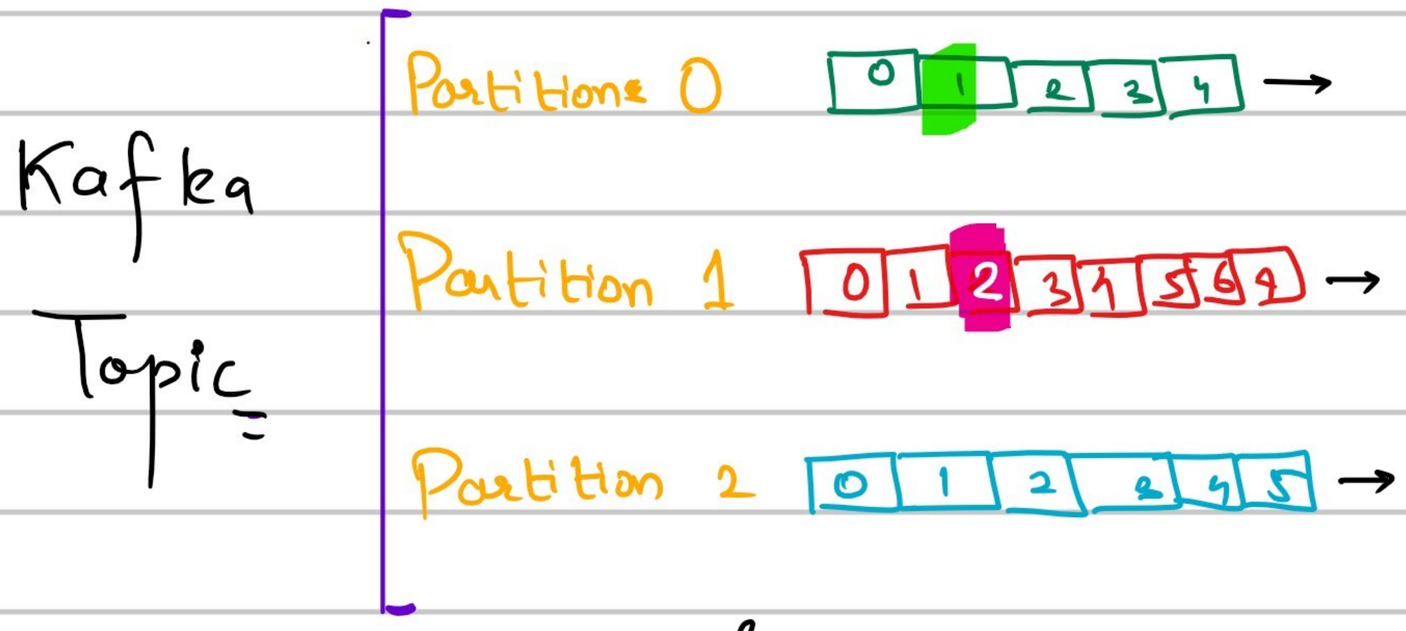


* Each truck sends the GPS position to Kafka.

* Data is sent to the topic → truck_gps

* Each truck sends a message after 20 seconds, each message contains truck_id & the truck position (latitude & longitude)

Offsets



* Offsets have meaning only for a specific partition

* Order is guaranteed only within a partition (not across the partition)

* Data is kept for 1 week (configurable)

* Data once written to the partition it cannot be changed (immutable)

* Data is assigned to the partition randomly, unless the key is provided

* Brokers & Topics

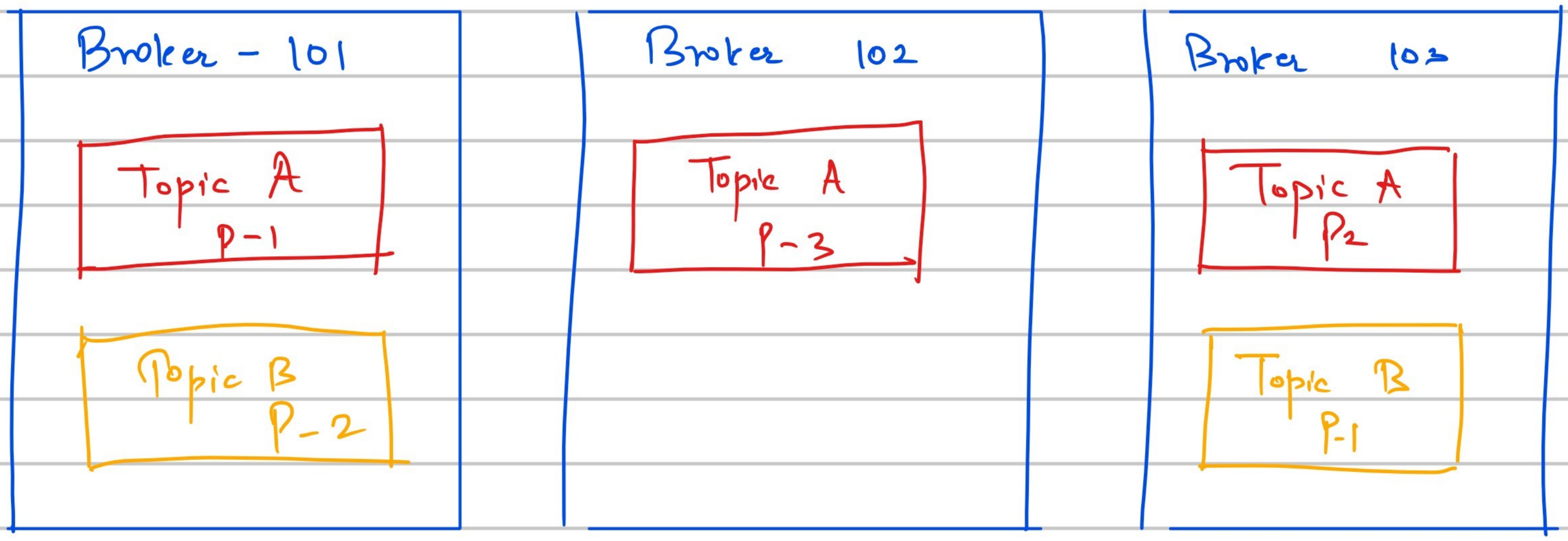
→ A Kafka cluster is composed of multiple brokers (servers)

→ Each broker is identified with its ID → (integer)

→ Each broker contains certain topic partitions.

→ We connect to any broker → Bootstrap Broker

No of brokers → 3
Topic A → 3 partitions
Topic B → 2 partitions



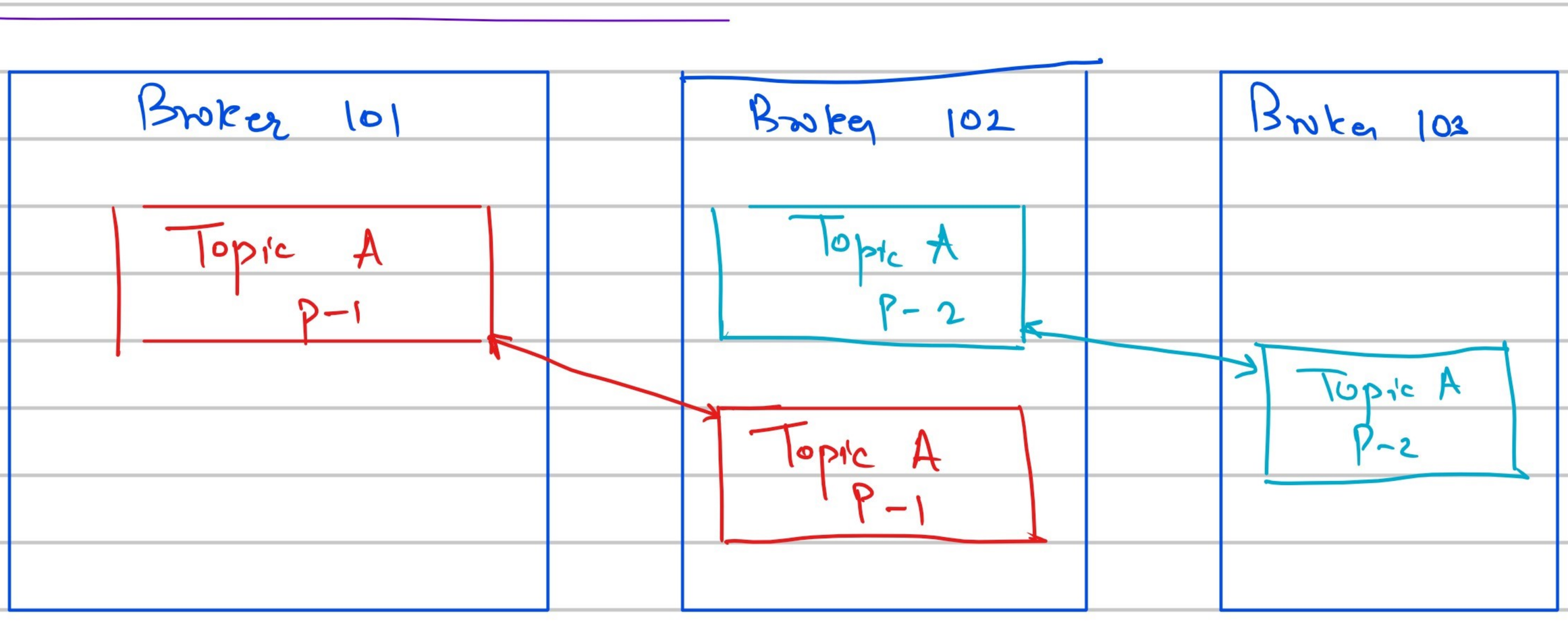
* Topic Replication factor

A topic should have Replication factor greater than 1 (usually between 2 & 3)

* Replication Factor make sures that if a broker is down; another broker can serve the data.

RF-2 [1 original + 1 copy]

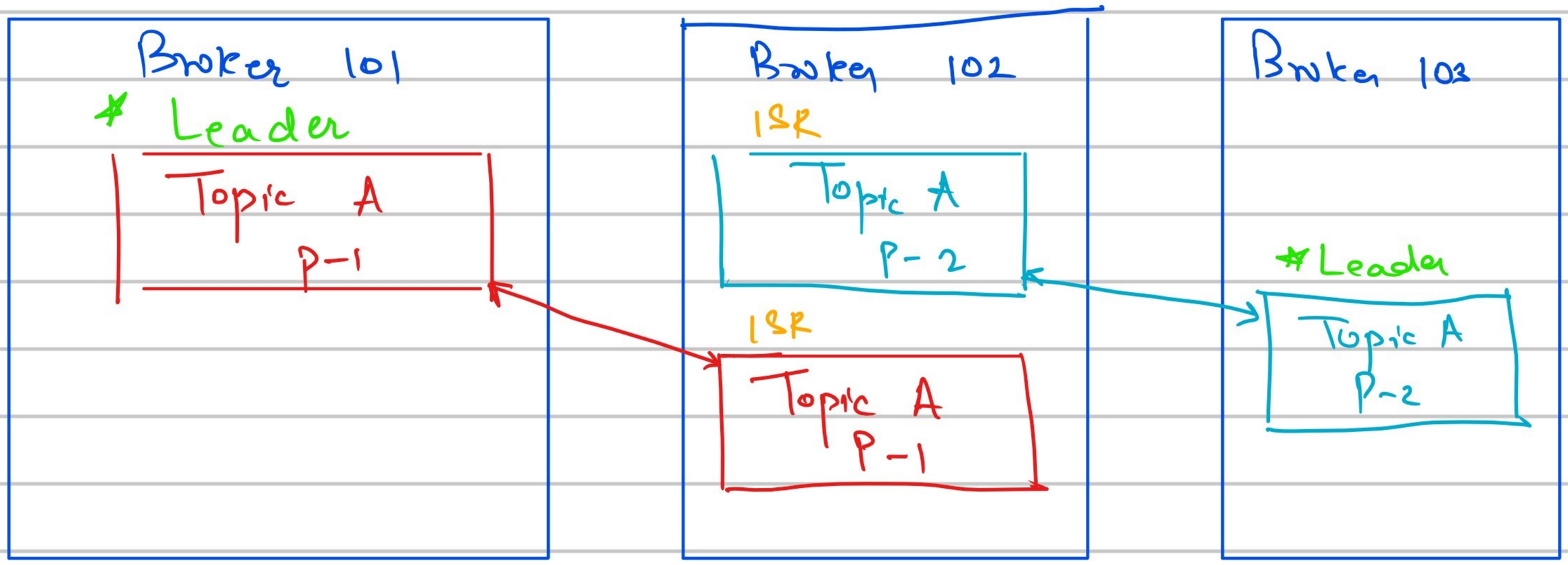
Topic A → 2 partition & RF → 2



* Concept of Leader for a Partition

[At any point in time, only one BROKER can be a LEADER for a PARTITION]

[ONLY that LEADER can RECIEVE & SERVE DATA for that Partition]



* Other brokers will synchronize the data

* Each partition has ONE LEADER & Multiple ISR

[In Sync Replicas]

RF -> 5

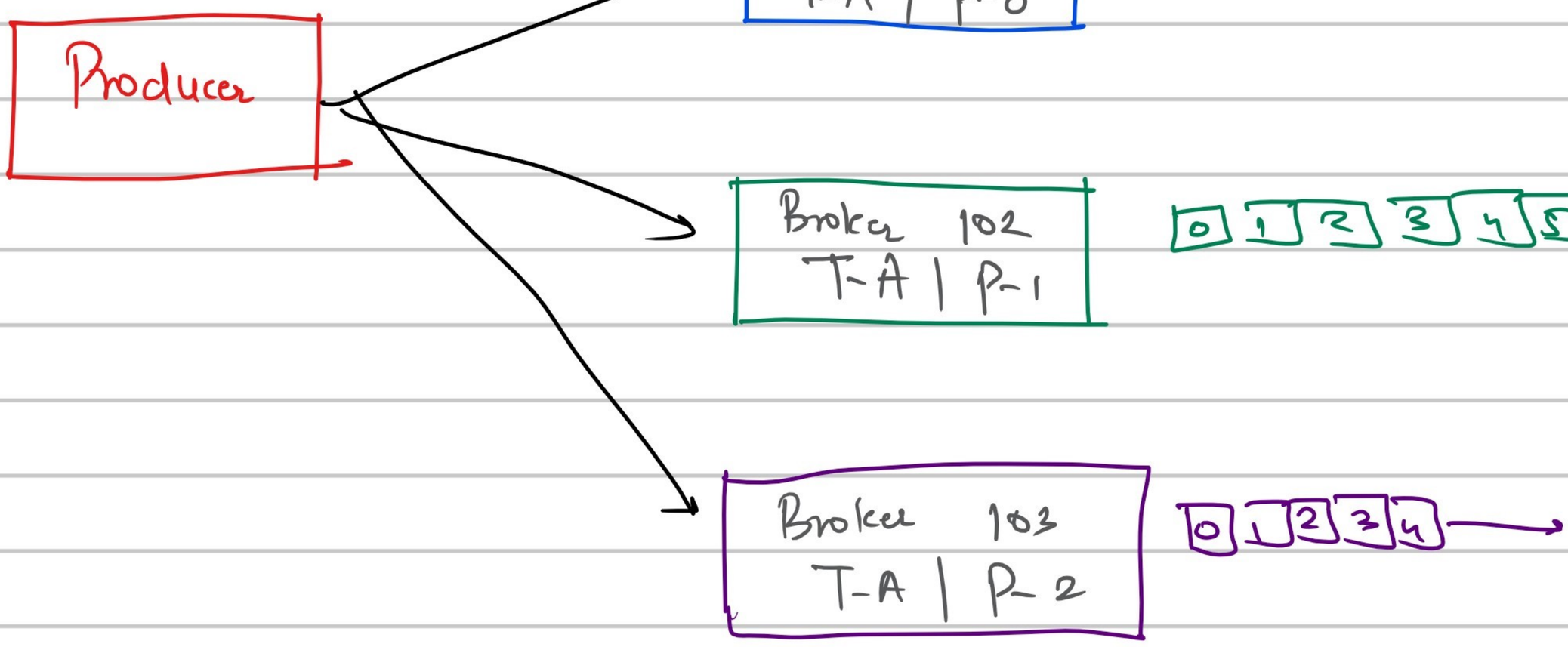
1 Leader

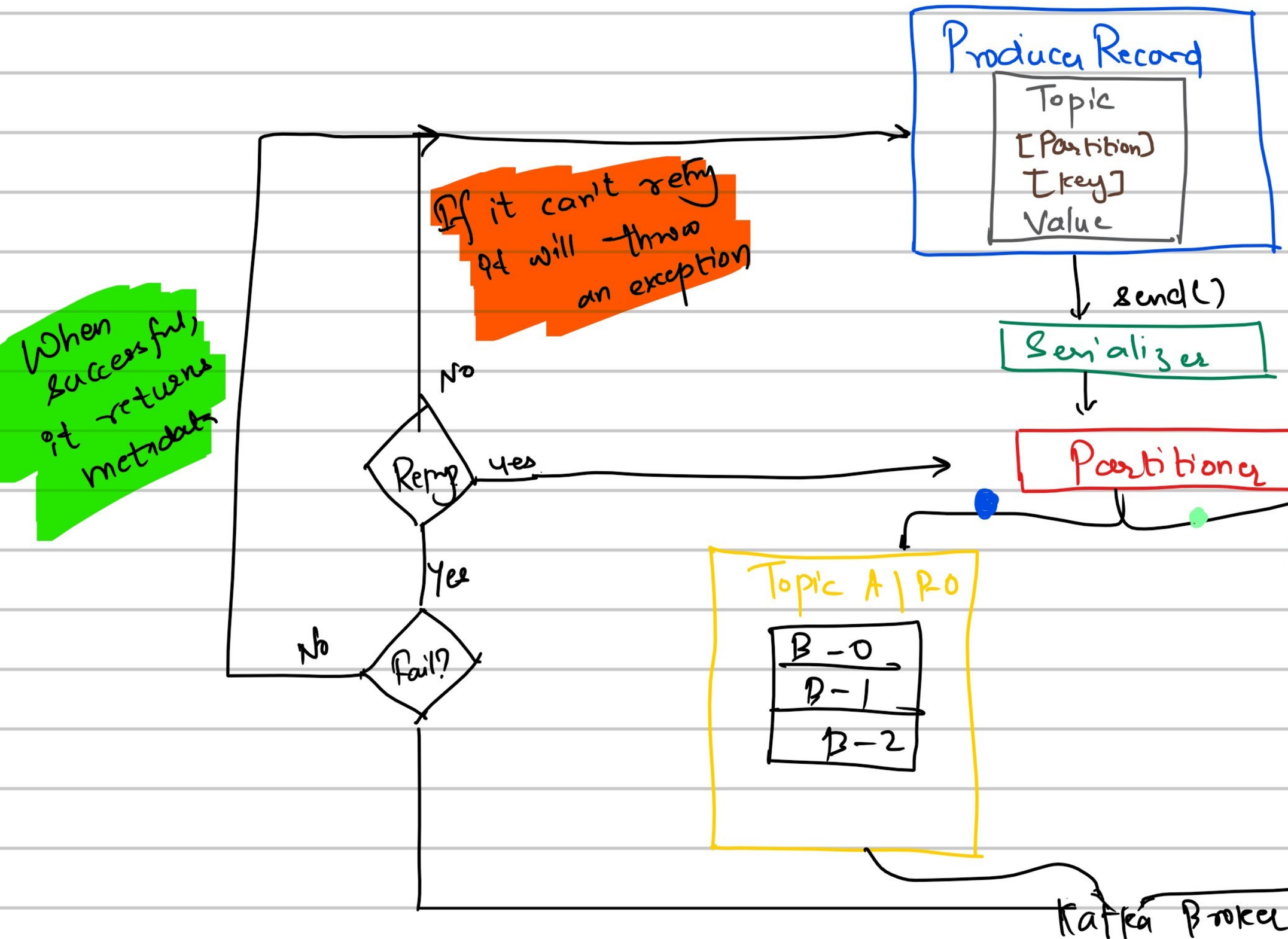
4 ISR

* Producers *

- Producer writes the data to the topics
- Producers automatically knows to which broker & partition to write to.

  * In case a Broker fails; Producer will automatically recover →





* Producers can choose to recieve the acknowledgement of data writes *

acks = 0 | Producer will not wait for acknowledgement (possible data loss)

acks = 1 | Producer will wait for 'Leader's Acknowledgement' (limited data loss)

acks = all | Leader + Replica's Acknowledgement (no data loss)

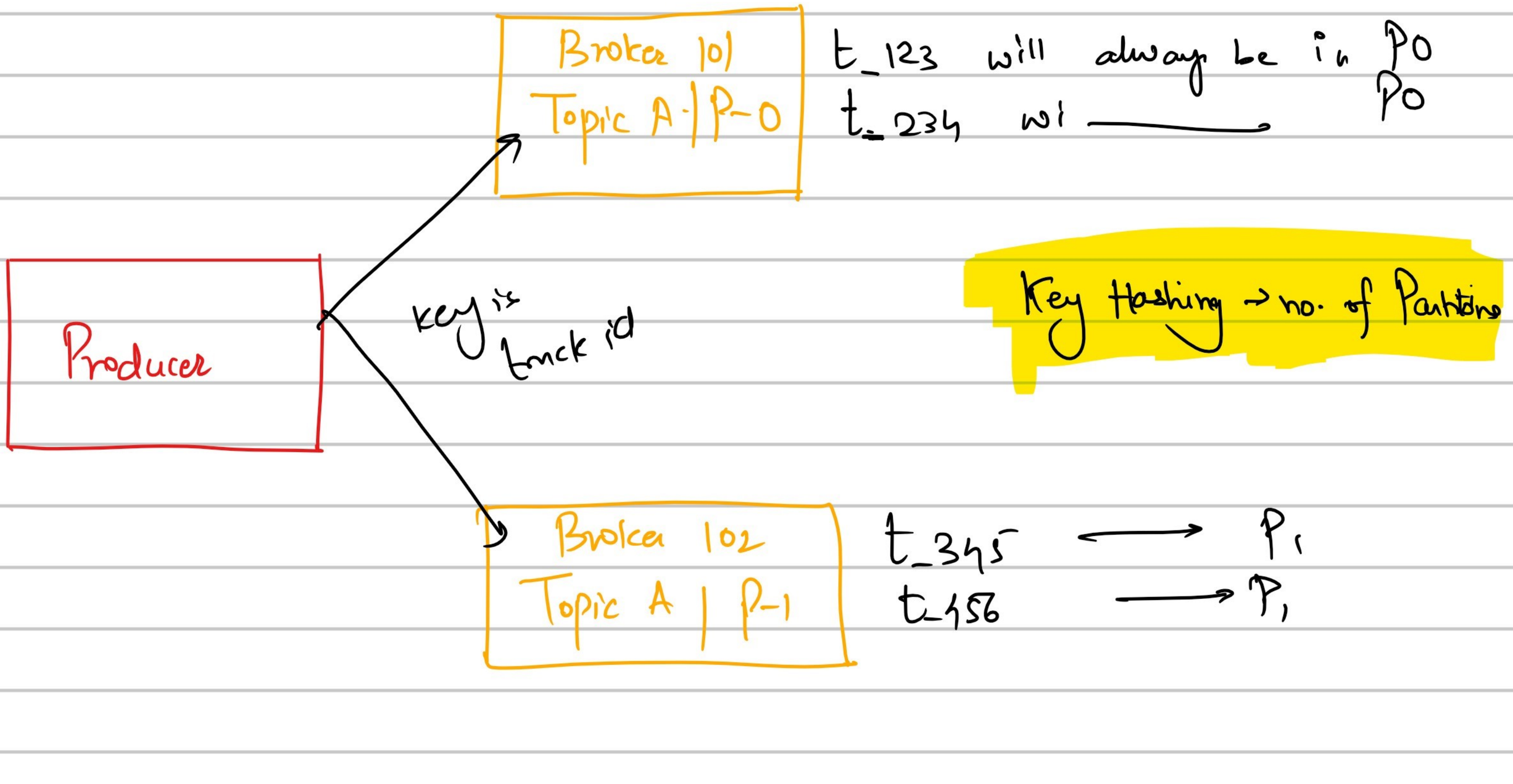
* Producer → Message Keys

- Producer can choose to send a key with the message (string, number, etc).

- If key=null; data is sent in Round Robin (101-102-103)

- If key is sent; then all the messages for that particular key will always land up in same partition.

→ A key is normally sent with the message when we need ordering for a specific field ex. truck_id



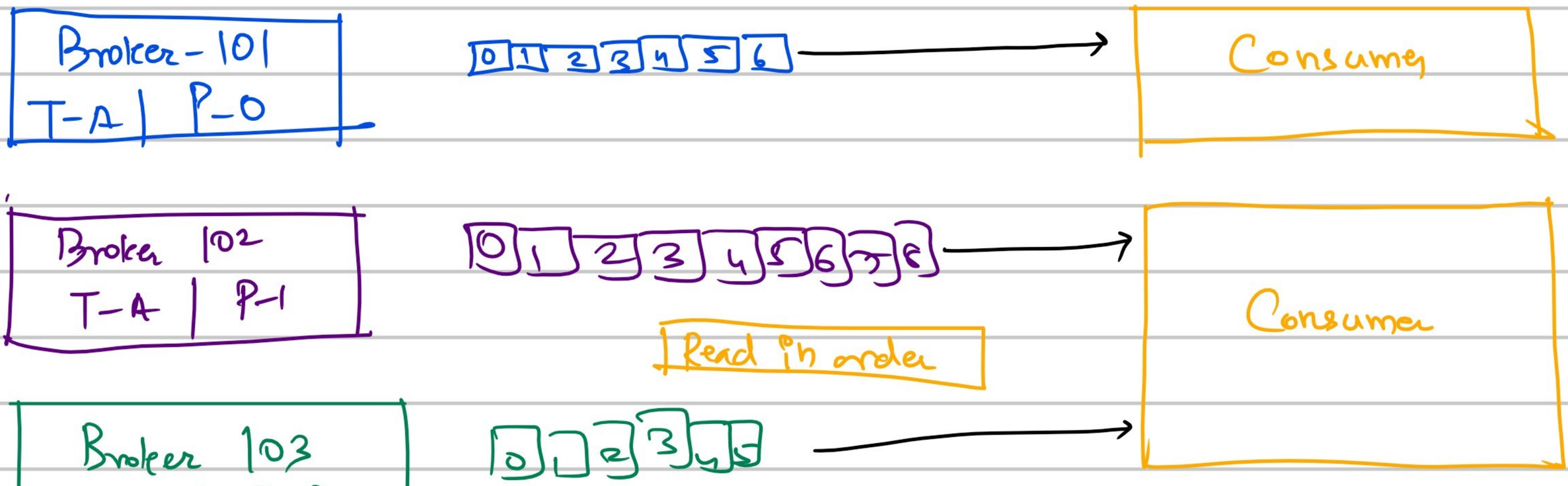
* Consumer *

→ Consumers read the data from a topic

* Consumer knows which broker to read from *

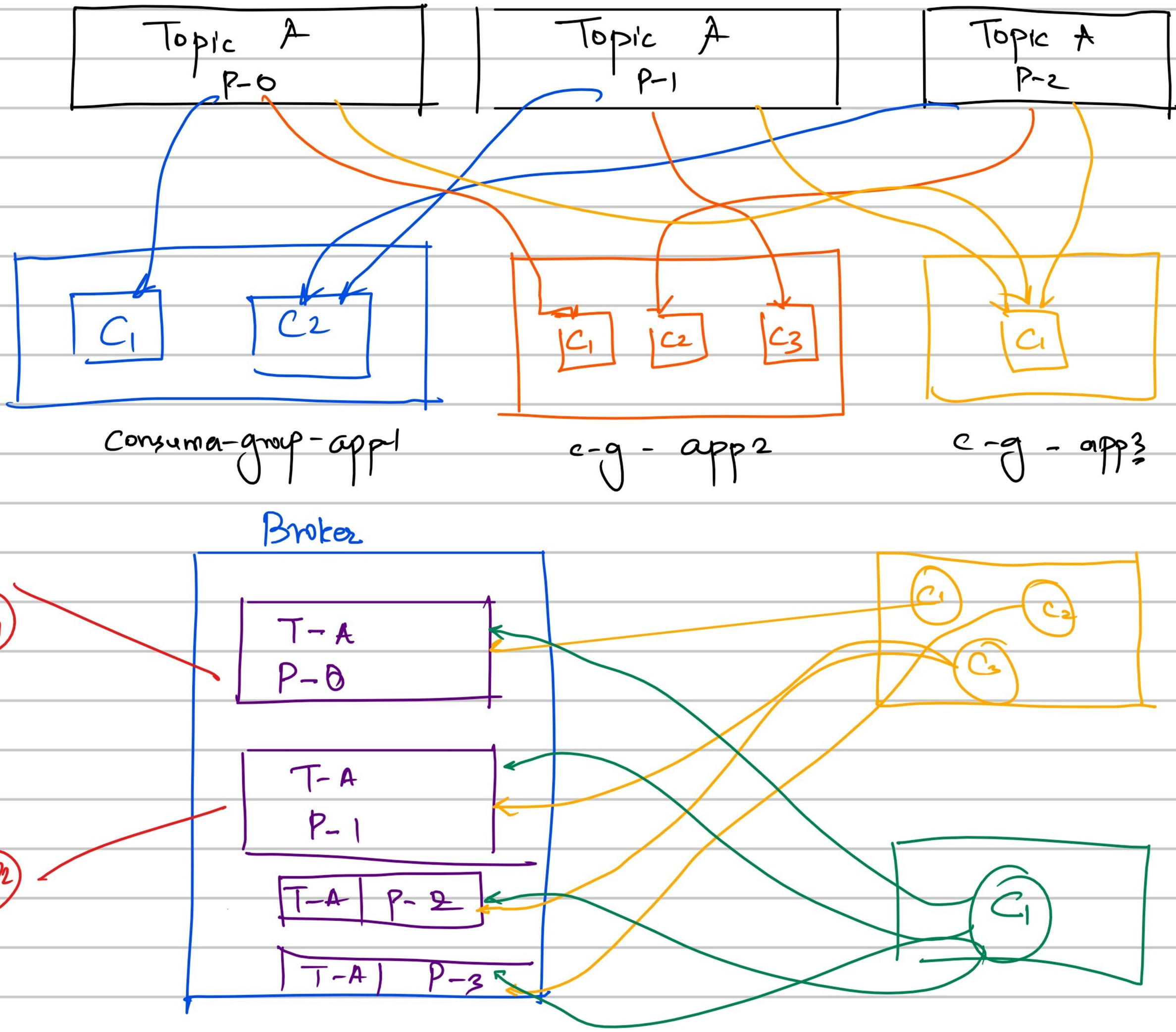
* In case of broker failure; consumer knows how to recover.

Data is read from each partition in an order



* Consumer Group *

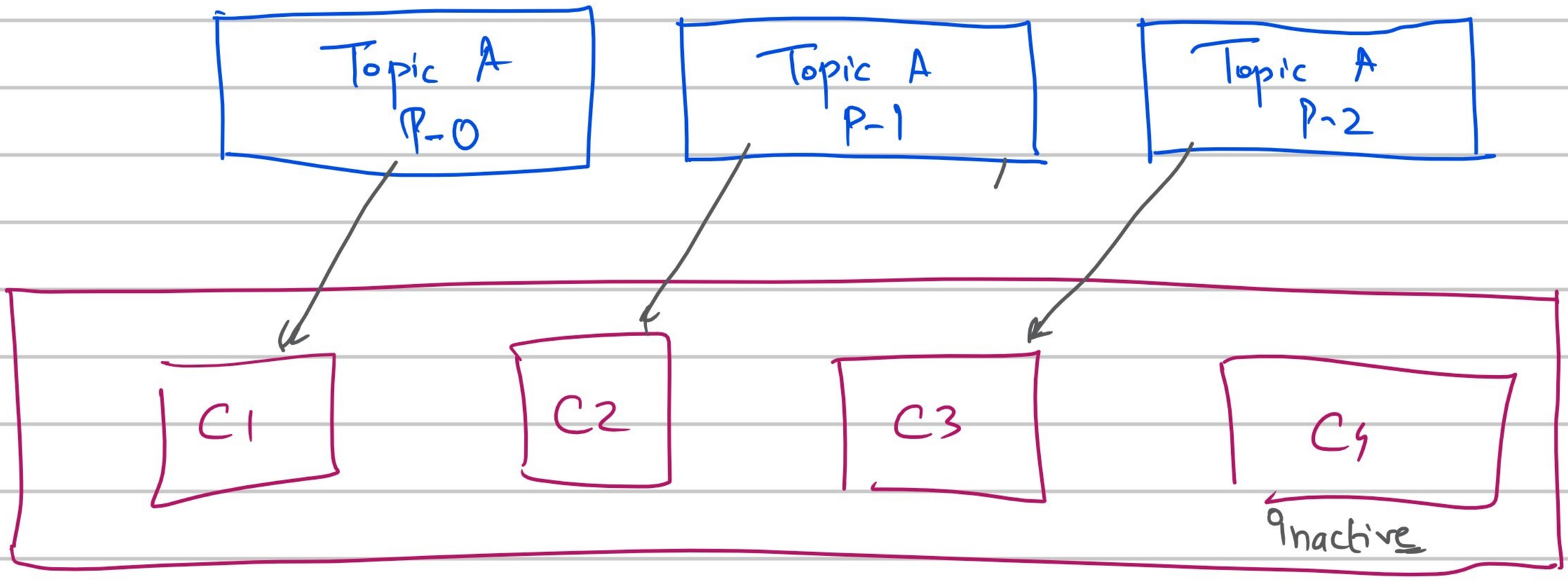
- Consumer reads the data in Consumer groups.
- Each consumer within a group reads from a exclusive partition
- If you have more consumers than the partitions; some consumers will be inactive



* Consumer belonging to the same group shares a group_id

Consumers in the group divides the topic's partition fairly amongst themselves → Each partition is only consumed by a single consumer in that group

* What will happen if number of consumers are more than number of partitions?



* Rebalancing *

→ when a consumer group scales up or scales down

Consumers in the group split the partition amongst themselves.

* Rebalancing is triggered by a shift in ownership between the partition & consumer

$CG \rightarrow (C_1, C_2, C_3) \rightarrow C_2$ suffers a failure.

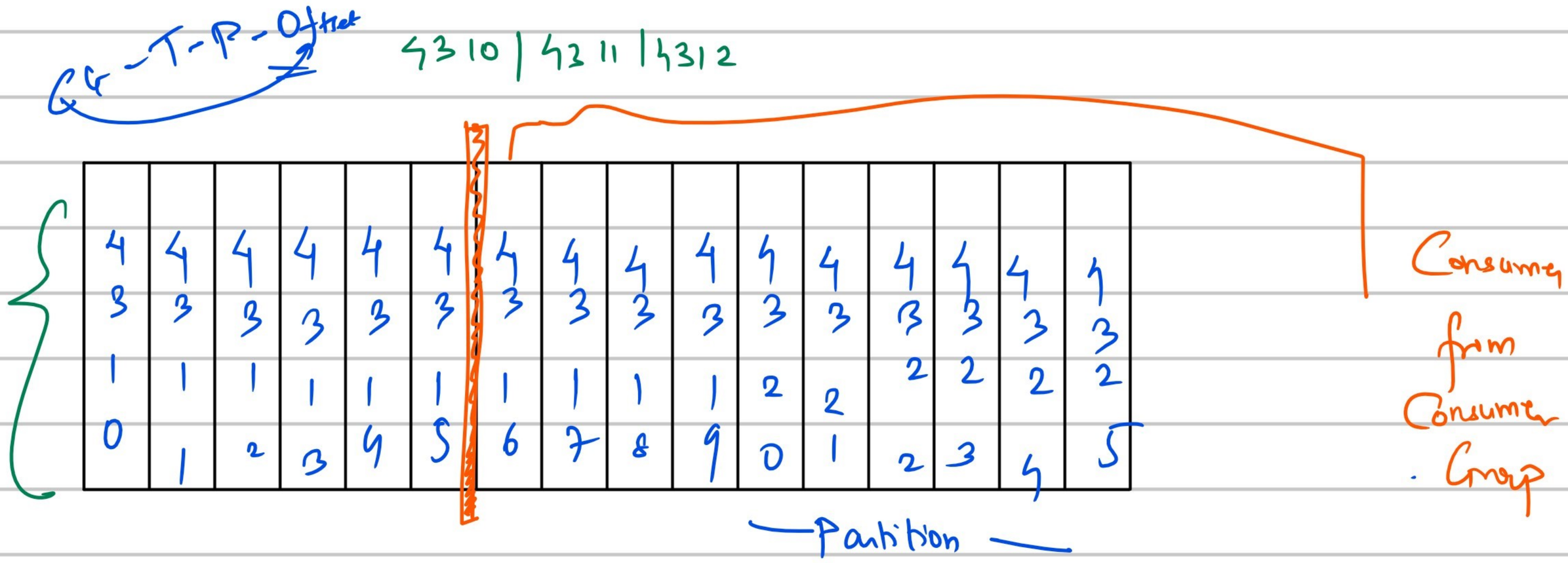
C_1 & C_3 will briefly pause consumption of messages from their partitions & all the partitions will be up for reassignment.

* Consumer Offsets *

→ Kafka stores the offset at which consumer group has been reading.

→ The offset committed → they are stored in a Kafka topic __consumer_offset

* When a Consumer group have processed the data recieved from Kafka ; it commits the offset



If a consumer is down ;
it will be able to read from where it left

* Delivery Semantics for Consumer *

* Consumer can choose when to commit offset.
* There are 3 delivery semantics

* At most Once
  → Offset is commited as soon as the message is recieved.
  → If processing goes wrong ; message will be lost.

* At Least Once → Usually Preffered.

→ Offset is commited after the message is processed

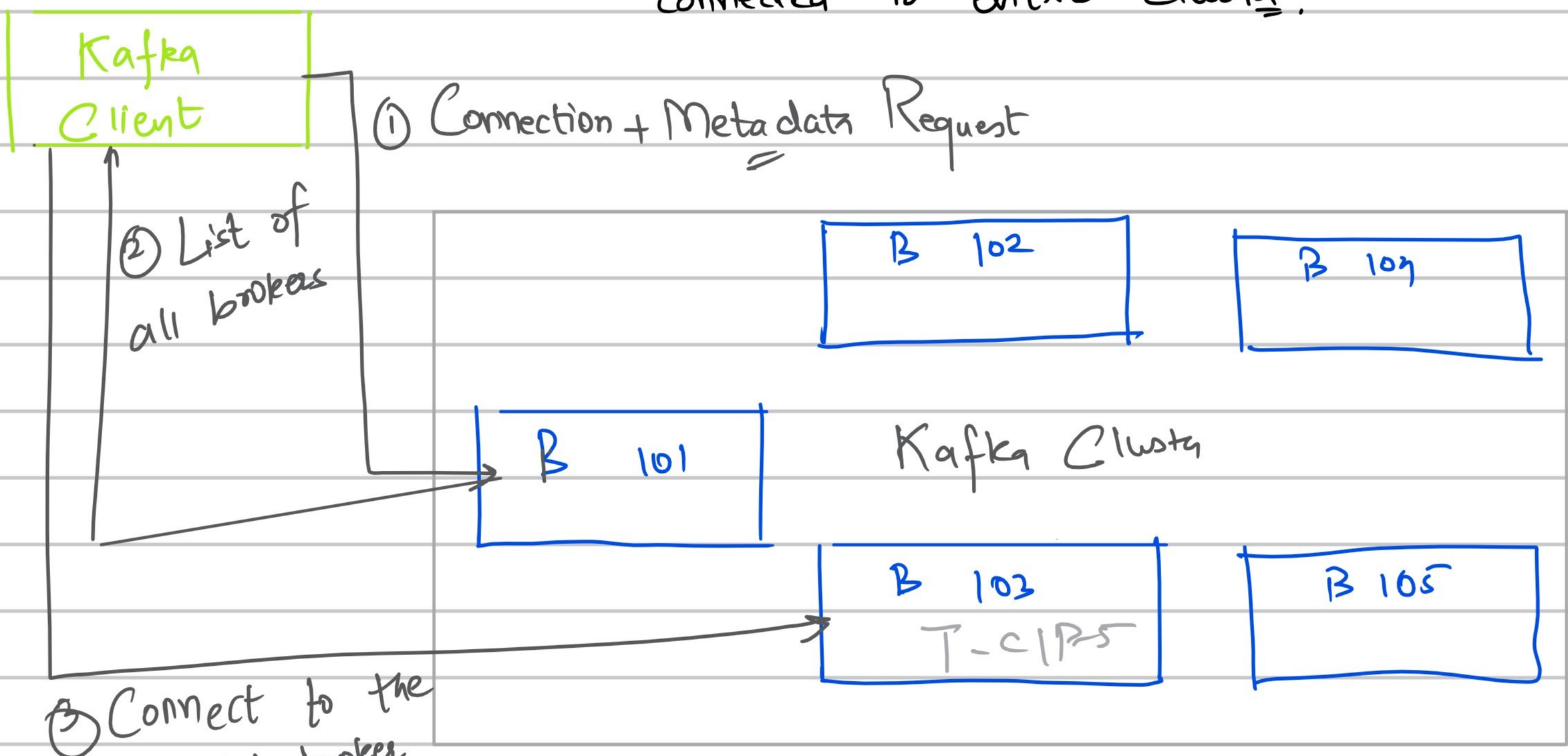
→ If the processing goes wrong; message will be read again

{ Idempotent } → if process is done multiple time it doesn't affect the state of application

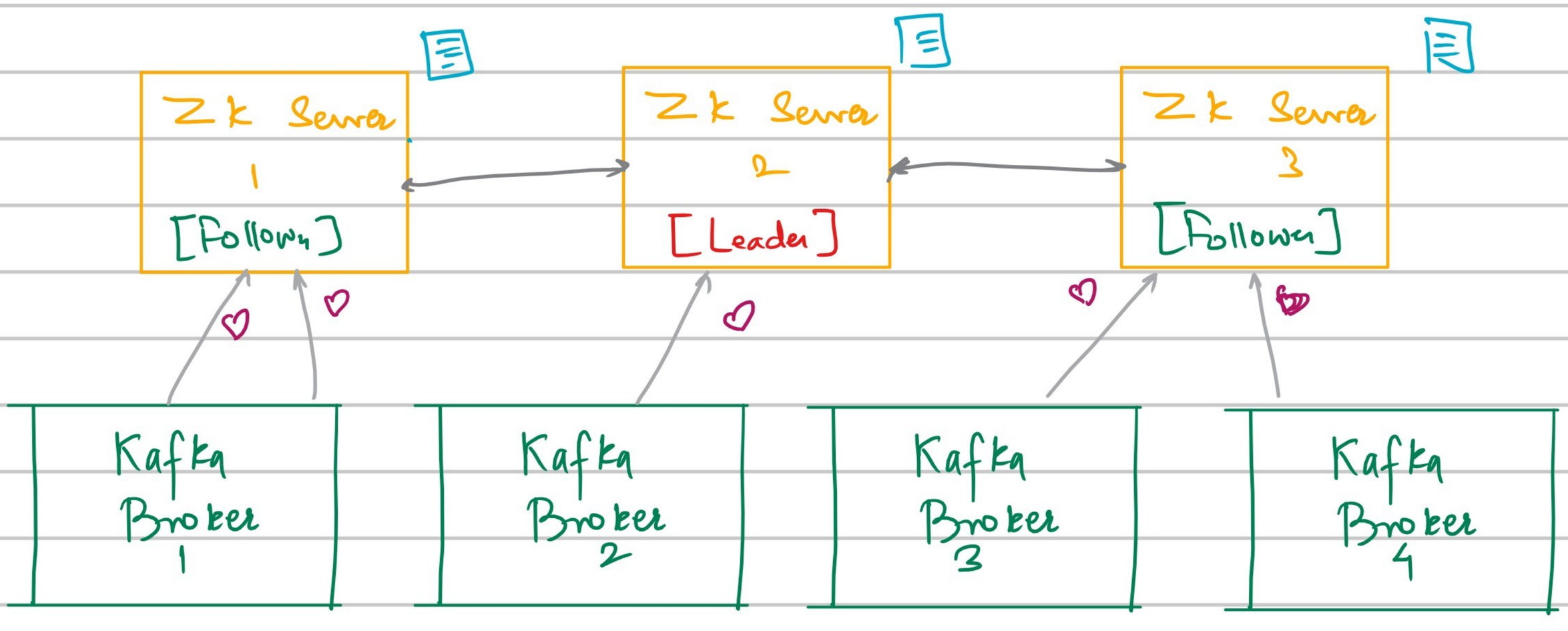
** Kafka Broker Discovery **

→ Every Kafka Broker is called as 'Bootstrap Server'

- You need to connect to any one broker & you will be connected to entire cluster.



* Zookeeper
  - → Zookeeper manages the brokers (keeps a list of them)
  - → Zookeeper helps in performing leader elections for partitions
  - → Zookeeper sends notification to Kafka in case of
    - New topic
    - Broker dies
    - → Broker comes up
    - Deleted Topic
    - etc
  - Zookeeper operates with odd number of servers [3, 5, 7 --]
  - Zookeeper has a leader (handle write) & rest of the servers are followers (handle read)

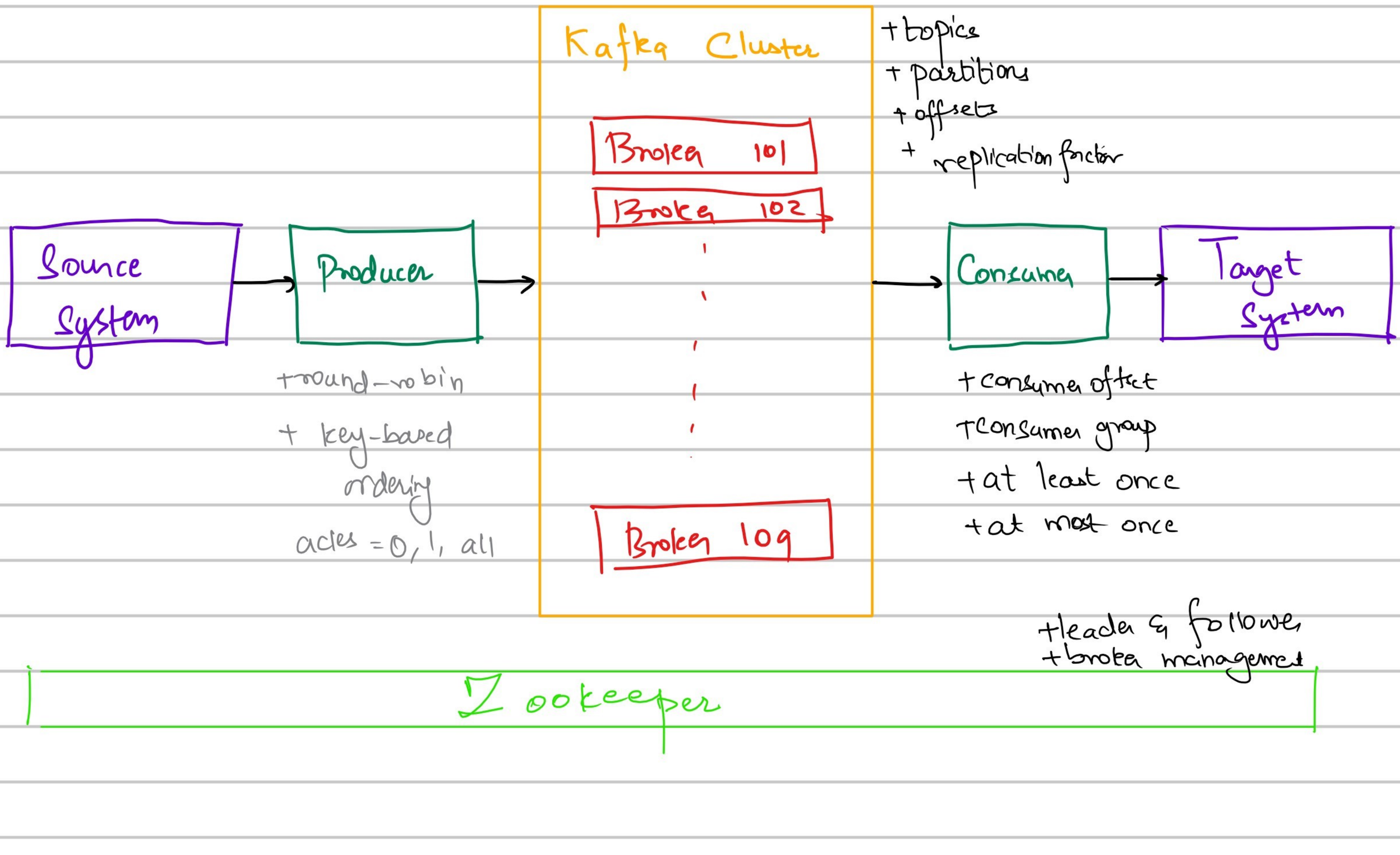


* * * Kafka Guarantees * * *

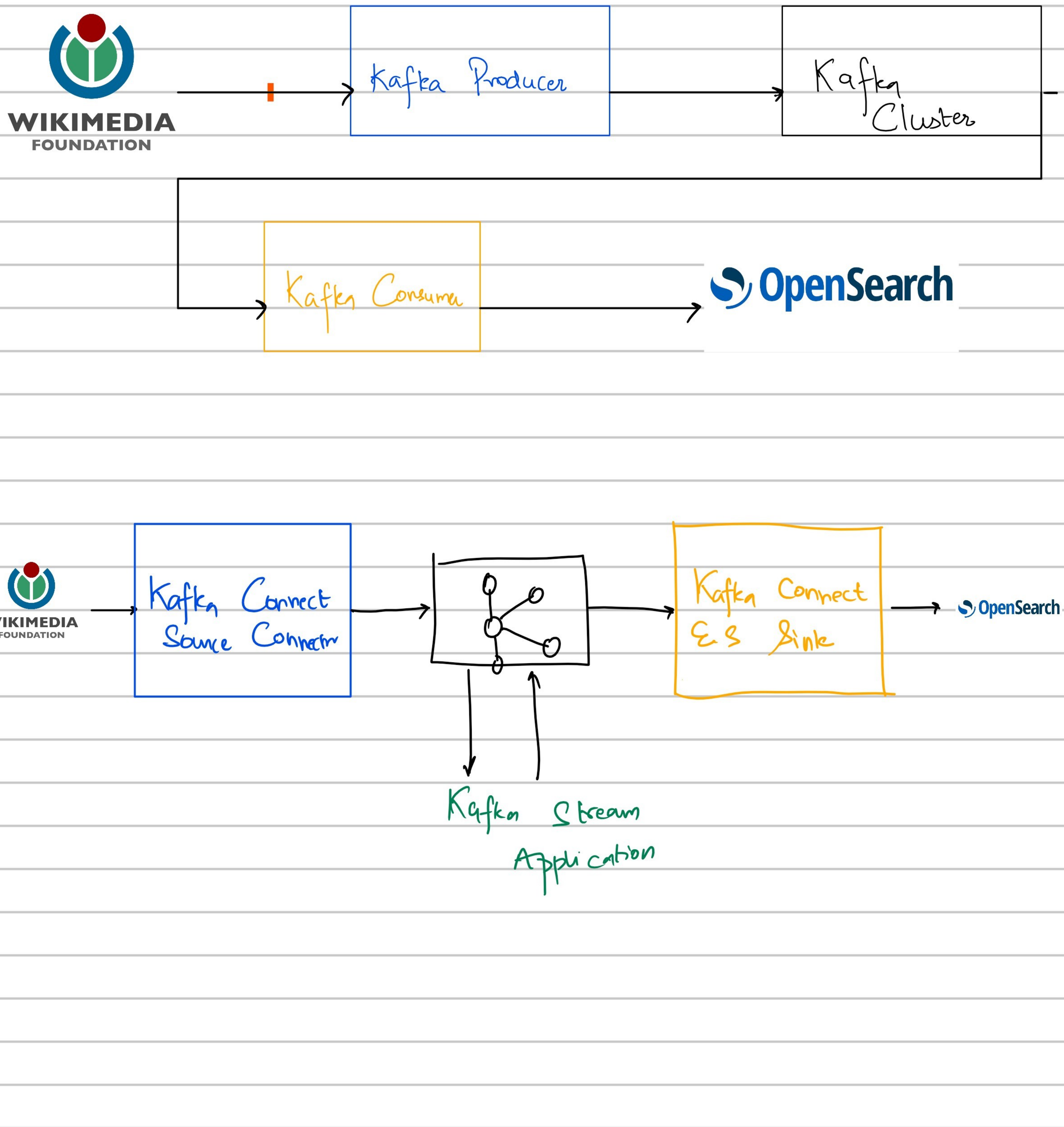
- Messages are appended to a topic-partition in order they are sent.
- Consumers reads messages in order stored in topic-partition.
- With RF → N ; producers and consumers can tolerated N-1 brokers being going down.

3 - 1 = 2

- RF - 3 is a good idea
- As long as number of partitions remains constant; same key will go to same partition



Case Study


WIKIMEDIA FOUNDATION
Kafka Producer
Kafka Cluster
Kafka Consumer
OpenSearch
WIKIMEDIA FOUNDATION
Kafka Connect Source Connector
Kafka Connect ES Sink
OpenSearch
Kafka Stream Application

Producer Configurations

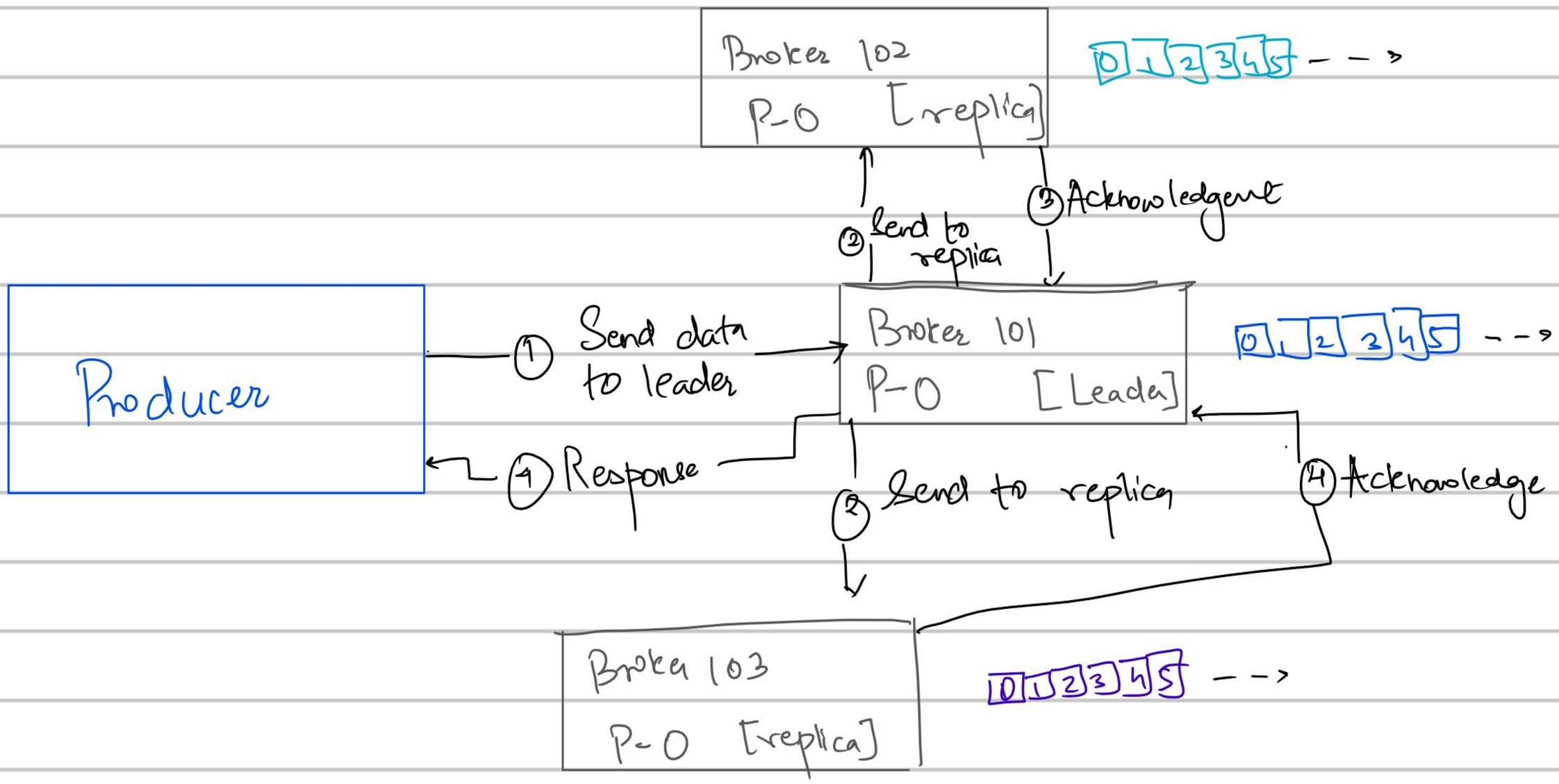
acks → Producer Acknowledgement

↳ 0, 1, all (-1)

0 → Producer wont wait for acknowledgement

1 → Producer will wait for Leader Acknowledgement

all → Leader + Replicas



Producer acks=all & min.insync.replicas

---

min.insync.replicas

When a producer sets acks to "all" (or "-1"), min.insync.replicas specifies the minimum number of replicas that must acknowledge a write for the write to be considered successful. If this minimum cannot be met, then the producer will raise an exception (either NotEnoughReplicas or NotEnoughReplicasAfterAppend).

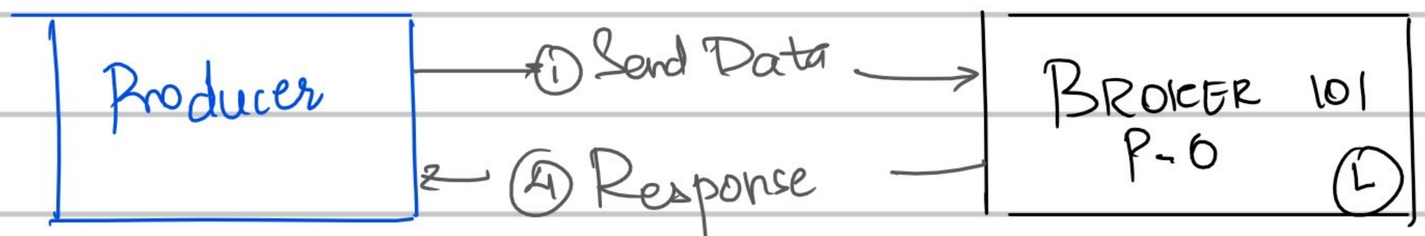
When used together, min.insync.replicas and acks allow you to enforce greater durability guarantees. A typical scenario would be to create a topic with a replication factor of 3, set min.insync.replicas to 2, and produce with acks of "all". This will ensure that the producer raises an exception if a majority of replicas do not receive a write.

Type:	int
Default:	1
Valid Values:	[1,...]
Importance:	high
Update Mode:	cluster-wide

acks=all

acks = all

min.insync.replicas = 2 → at least LEADER BROKER & ONE REPLICA NEEDS TO ACK



KAFKA TOPIC AVAILABILITY

RF = 3

acks = 0, 1 → if one partition is up & considered an ISR : the topic will be available for writing

acks = all

min.insync.replicas = 1

A topic must have atleast 1 partition up as an ISR (including Leader) → tolerate 2 brokers being down

min.insync.replicas = 2

A topic should atleast have 2 ISR up

→ we can tolerate at most one broker being down

→ It gives us a guarantee that every write, data will be written at least twice

min.insync.replicas = 3 → We can't tolerate any broker down

RF = N

min.insync.replicas = M

acks = all

Broker Down Tolerated → N - M

* Producer Retries.

→ In case of transient failures, developers are expected to handle ~~expect~~ exceptions, otherwise data will be lost

Example of transient failure → NOT_ENOUGH_REPLICAS

* There is a retries setting

Kafka 2.0 =< → 0

Kafka 2.1 >= → 2147483647

retry.backoff.ms → 100 ms

retries

Setting a value greater than zero will cause the client to resend any record whose send fails with a potentially transient error. Note that this retry is no different than if the client resent the record upon receiving the error. Allowing retries without setting `max.in.flight.requests.per.connection` to 1 will potentially change the ordering of records because if two batches are sent to a single partition, and the first fails and is retried but the second succeeds, then the records in the second batch may appear first. Note additionally that produce requests will be failed before the number of retries has been exhausted if the timeout configured by `delivery.timeout.ms` expires first before successful acknowledgement. Users should generally prefer to leave this config unset and instead use `delivery.timeout.ms` to control retry behavior.

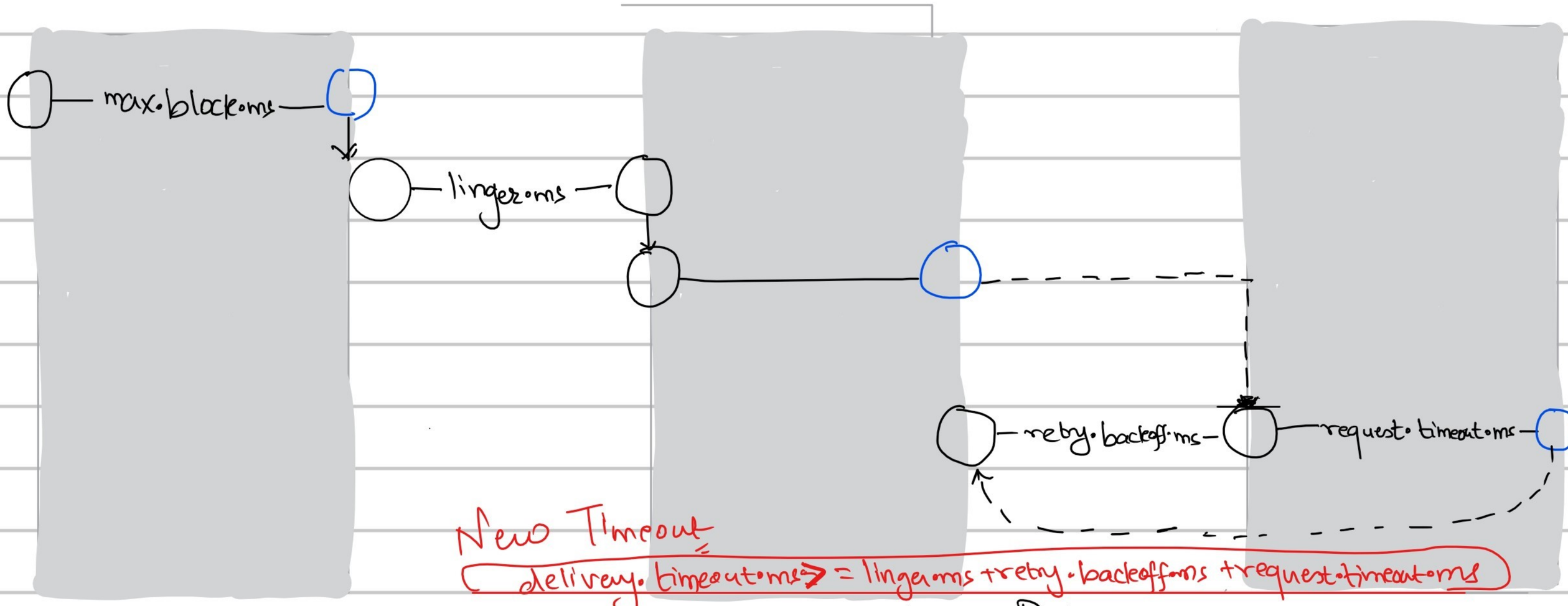
Type: int
Default: 2147483647
Valid Values: [0,...,2147483647]
Importance: high

retry.backoff.ms

The amount of time to wait before attempting to retry a failed request to a given topic partition. This avoids repeatedly sending requests in a tight loop under some failure scenarios.

Type: long
Default: 100
Valid Values: [0,...]
Importance: low

Producer Timeout.



If retries > 0 → retries = 2147483647, retries are bounded by timeout

Since - kafka 2.1 delivery.timeout.ms = 120000 ~ 2 mins

request.timeout.ms

The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is not received before the timeout elapses the client will resend the request if necessary or fail the request if retries are exhausted.

Type: int
Default: 30000 (30 seconds)
Valid Values:
Importance: high
Update Mode: read-only

max.block.ms

The configuration controls how long the `KafkaProducer` 's `send()` , `partitionsFor()` , `initTransactions()` , `sendOffsetsToTransaction()` , `commitTransaction()` and `abortTransaction()` methods will block. For `send()` this timeout bounds the total time waiting for both metadata fetch and buffer allocation (blocking in the user-supplied serializers or partitioner is not counted against this timeout). For `partitionsFor()` this timeout bounds the time spent waiting for metadata if it is unavailable. The transaction-related methods always block, but may timeout if the transaction coordinator could not be discovered or did not respond within the timeout.

Type: long
Default: 60000 (1 minute)
Valid Values: [0,...]
Importance: medium

Records will be failed if they can't be acknowledged within delivery.timeout.ms

Producer Retries: Warning for Old Kafka Versions

If we are using non-Idempotent producer.

In case of retries → there is a chance that messages will be out of order

→ If we rely on key based ordering; that can be an issue.

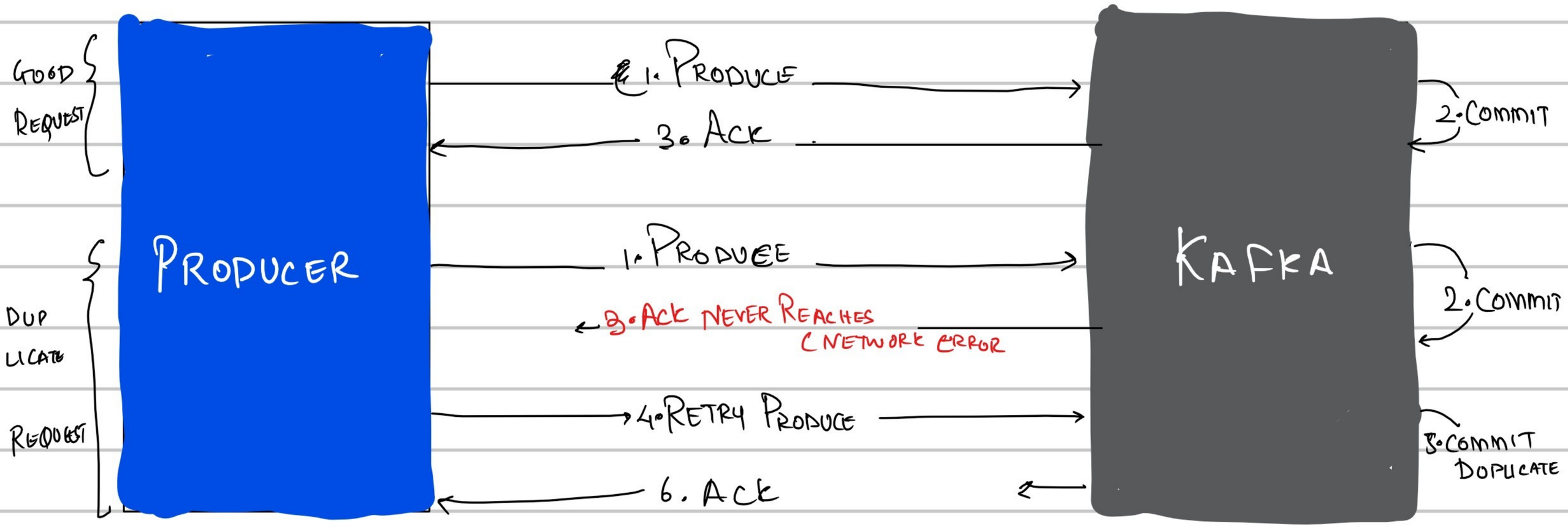
For solution → set the number of produce requests can be made in parallel

max.in.flight.requests.per.connection

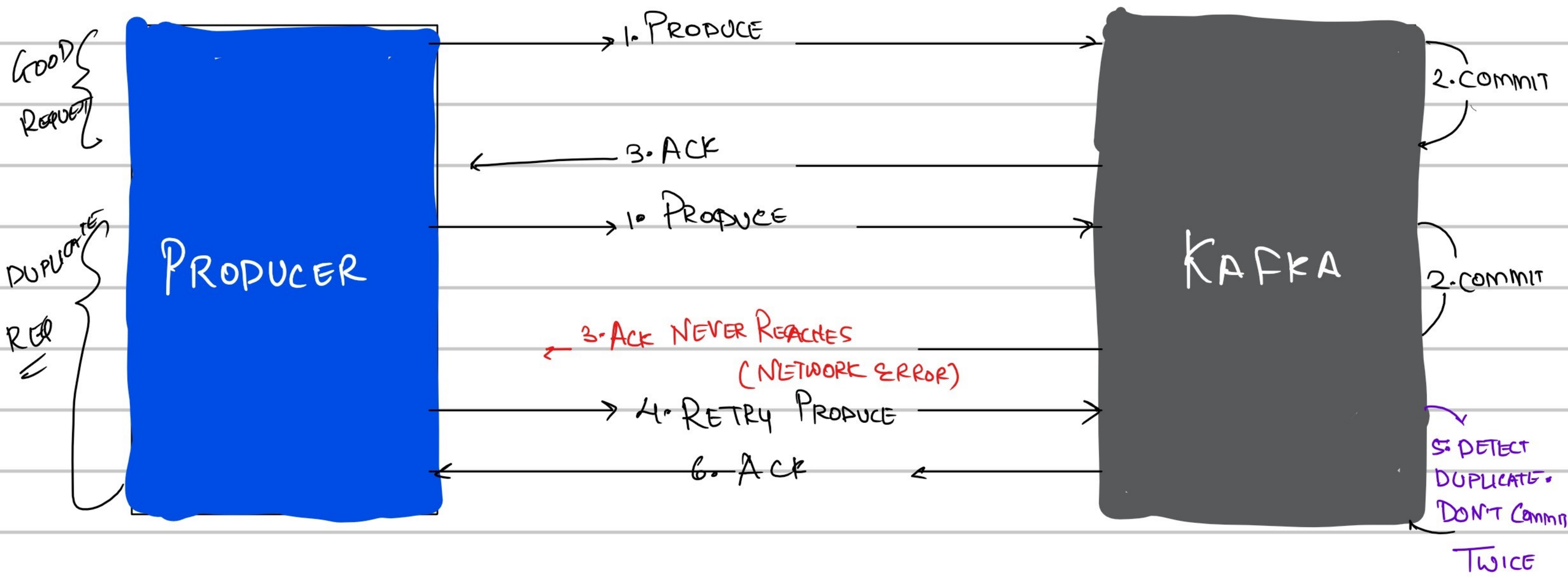
The maximum number of unacknowledged requests the client will send on a single connection before blocking. Note that if this config is set to be greater than 1 and `enable.idempotence` is set to false, there is a risk of message re-ordering after a failed send due to retries (i.e., if retries are enabled).

Type: int
Default: 5
Valid Values: [1,...]
Importance: low

Idempotent Producer



Idempotent Producer → won't introduce duplicates on network error



Required Settings

↳ Idempotent Producers → guarantees stable & safe pipeline

from Kafka 3.0 → default

retries = Integer.MAX_VALUE - 2147483647

max.in.flight.requests = 5

acks = all

properties.setProperty("enable.idempotence", true);

In kafka 3-0 → producers are safe by default

acks=all → Make sure data is properly replicated before an ack is recieved.

min.insync.replicas=2 → Make sures that 2 brokers in ISR have data after an ack

enable.idempotence=true → Make sure → duplicate entries are not introduced due to network error.

retries=MAX_INT → Retry until delivery.timeout.ms is reached

delivery.timeout.ms = 120000 → Make sures fails after retrying for 2 mins

max.in.flight.requests.per.connection=5 → Make sure maximum performance while keeping message ordering

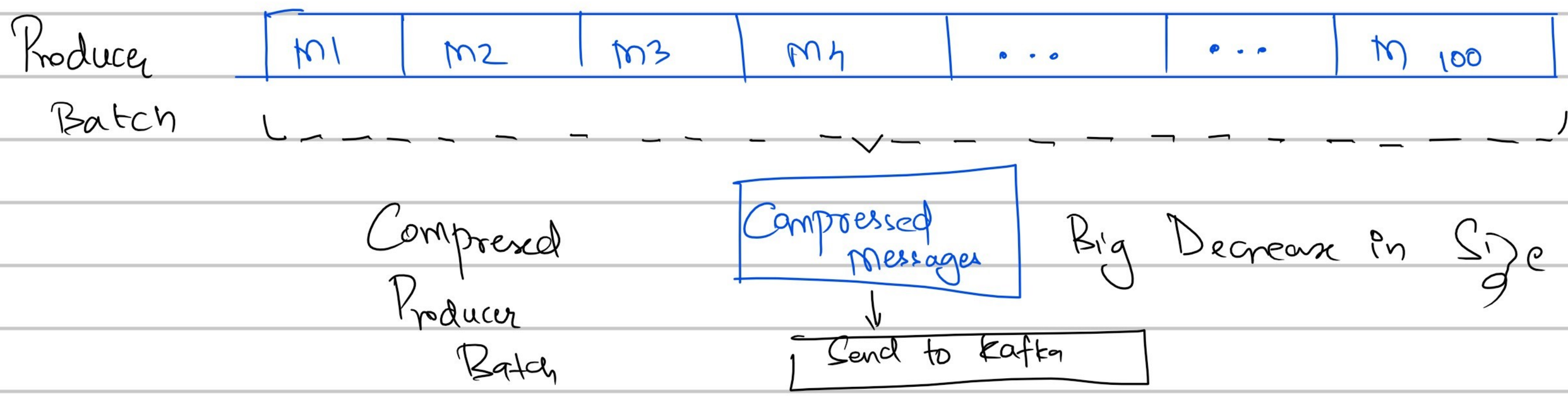
Message Compression → Producer Level

↝ Compression can be enabled at Producer Level

compression.type

- none (default)
- gzip
- lz4
- snappy
- zstd (kafka 2.1)

→ Compression is more effective the bigger the batch of messages being sent to kafka



Compression have advantages as well as disadvantages

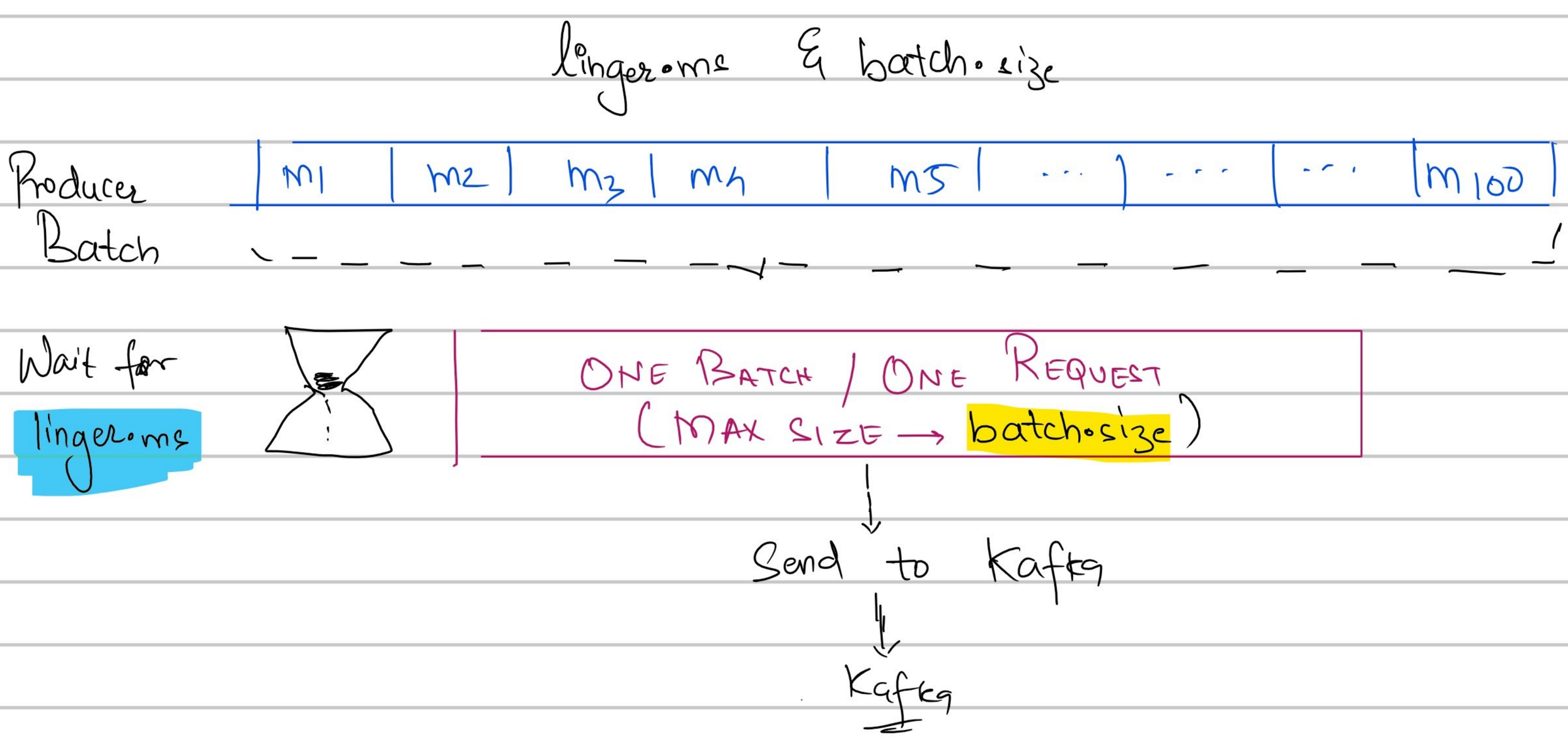
Adv
1. Smaller Producer Request Size.
2. Faster Transfer
3. Better Throughput
4. Better disk utilization

Disadvranges:
1. Producer must commit some CPU cycles to compression
2. Consumer must commit some CPU cycles to decompress

Overall

snappy lz4

linger.ms & batch.size to have bigger batches



linger.ms

The producer groups together any records that arrive in between request transmissions into a single batched request. Normally this occurs only under load when records arrive faster than they can be sent out. However in some circumstances the client may want to reduce the number of requests even under moderate load. This setting accomplishes this by adding a small amount of artificial delay—that is, rather than immediately sending out a record, the producer will wait for up to the given delay to allow other records to be sent so that the sends can be batched together. This can be thought of as analogous to Nagle's algorithm in TCP. This setting gives the upper bound on the delay for batching: once we get `batch.size` worth of records for a partition it will be sent immediately regardless of this setting, however if we have fewer than this many bytes accumulated for this partition we will 'linger' for the specified time waiting for more records to show up. This setting defaults to 0 (i.e. no delay). Setting `linger.ms=5`, for example, would have the effect of reducing the number of requests sent but would add up to 5ms of latency to records sent in the absence of load.

Type: long
Default: 0
Valid Values: [0,...]

batch.size

The producer will attempt to batch records together into fewer requests whenever multiple records are being sent to the same partition. This helps performance on both the client and the server. This configuration controls the default batch size in bytes.

No attempt will be made to batch records larger than this size.

Requests sent to brokers will contain multiple batches, one for each partition with data available to be sent.

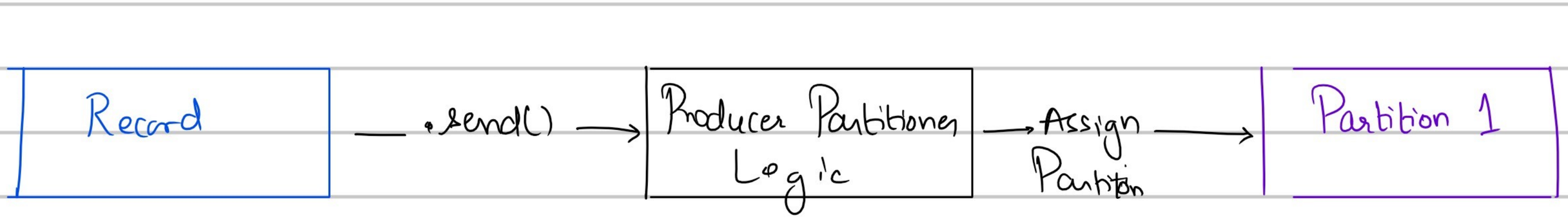
A small batch size will make batching less common and may reduce throughput (a batch size of zero will disable batching entirely). A very large batch size may use memory a bit more wastefully as we will always allocate a buffer of the specified batch size in anticipation of additional records.

A small batch size will make batching less common and may reduce throughput (a batch size of zero will disable batching entirely). A very large batch size may use memory a bit more wastefully as we will always allocate a buffer of the specified batch size in anticipation of additional records.

Note: This setting gives the upper bound of the batch size to be sent. If we have fewer than this many bytes accumulated for this partition, we will 'linger' for the `linger.ms` time waiting for more records to show up. This `linger.ms` setting defaults to 0, which means we'll immediately send out a record even the accumulated batch size is under this `batch.size` setting.

Type: int
Default: 16384
Valid Values: [0,...]
Importance: medium

Producer Default Partition when key != null



Key Hashing → process of determining the mapping of a key to partition

Murmur2 Algorithm

targetPartition = Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1)

Producer Default Partition when key = null

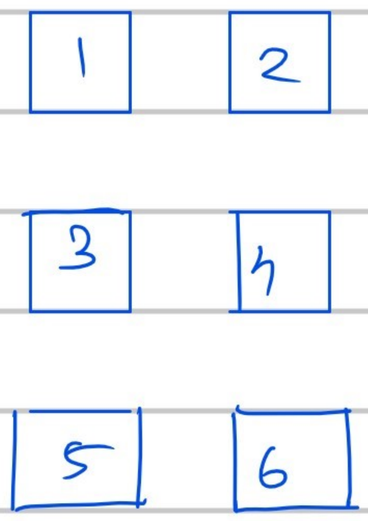
default partitioner

Round Robin → Kafka 2.3 & below

Sticky Partitioner → Kafka 2.4 & above.

Round Robin

This results in more batches & smaller batches

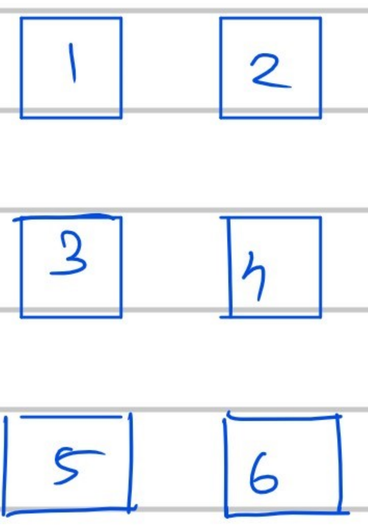


P_1	1 6
P_2	2
P_3	3
P_4	4
P_5	5

Sticky Partitioning

We stick to a partition until the batch is full or linger.ms has elapsed

After sending the batch, the partition that is sticky changes



P_1	1 2 3
P_2	4 5 6
P_3	
P_4	
P_5	

Larger batches & reduced latency

Over a period of time, records are evenly spread across partitions

max.block.ms & buffer.memory

* If the producer produces faster than broker can take, the records will buffer in memory.

buffer.memory = 33554432 (32 MB)

- The buffer will fill up over the time & empty back down when throughput to the broker increases

If the buffer is full (all 32 MB) then .send() will start to block

max.block.ms = 60000 → the send() will block until throwing an exception

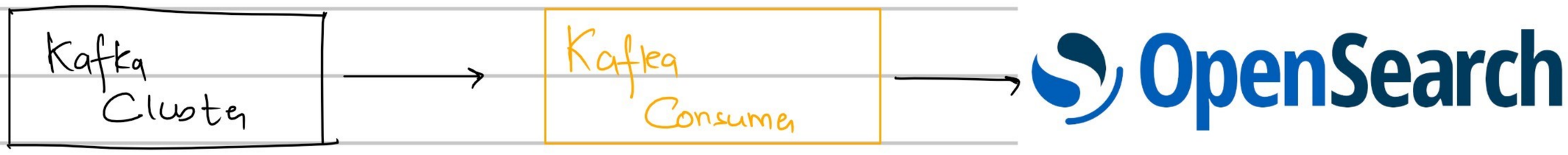
Exceptions are thrown when

① Producer has filled up the buffer.

② Broker is not accepting any new data

③ 60 secs has elapsed

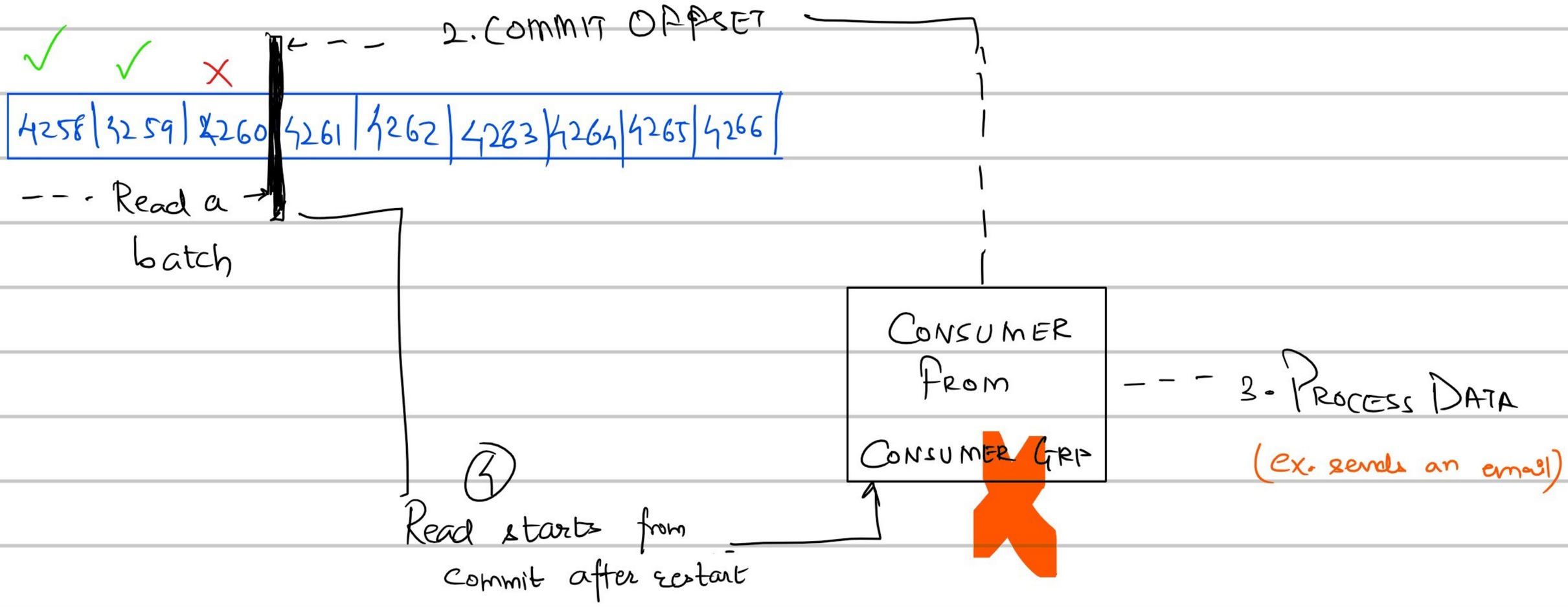
Consumer Implementation



bonsai.io

Delivery Semantics

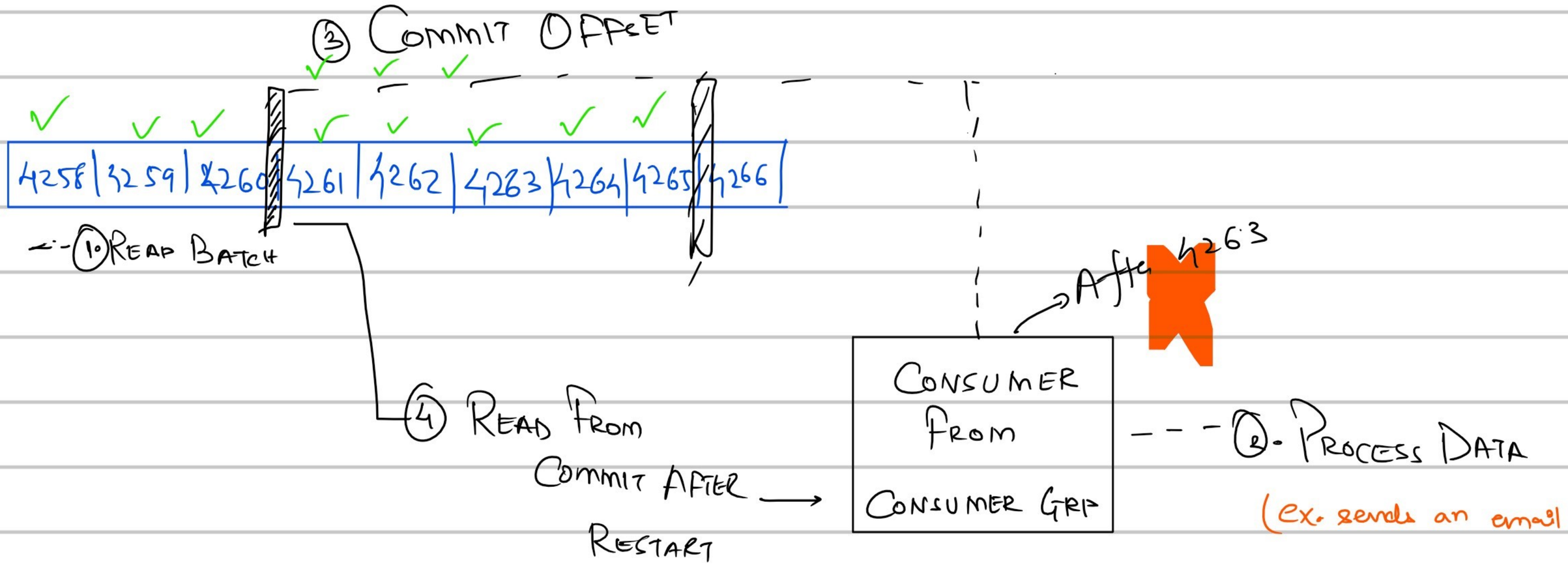
- At most once
  - ↳ offsets are committed as soon as the message batch is recieved. In case processing goes wrong; the message will be lost.



At least once.

↳ offsets are committed after the message is processed.

If processing goes wrong → message will be read again. This can result in duplicate processing of the message. → We need to make sure the processing is IDEMPOTENT



Consumer Offset Commit Strategies

There are two common strategies:

enable.auto.commit = true & synchronous processing of batches.

enable.auto.commit = false & manual commit the offset

Kafka Consumer → Auto Offset Commit Behavior

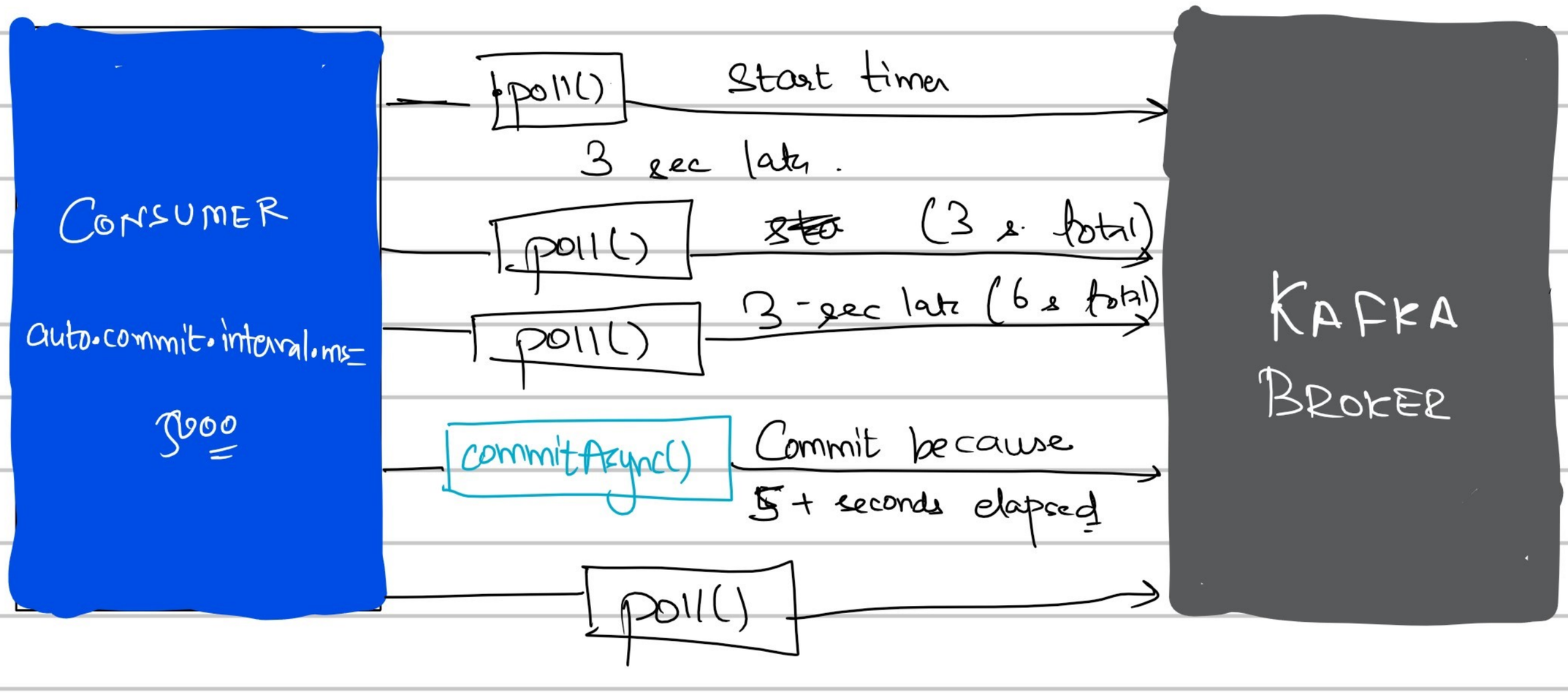
→ Java Consumer API → offsets are regularly commited

→ Enable at-least-once reading scenario by default

→ Offsets are committed when you call .poll() & auto.commit.interval.ms has elapsed

→ auto.commit.interval.ms = 5000 & enable.auto.commit = true

Make sure all the messages are processed before calling poll().



enable.auto.commit=true & synchronous process of batch

- With autocommit, offsets will be committed automatically at regular interval (auto.commit.interval.ms = 5000 by default) everytime you call .poll()

```
while(true){
    List<Records> batch=consumer.poll(Duration.ofMillis(100));
    doSomethingSynchronous(batch)
}
```

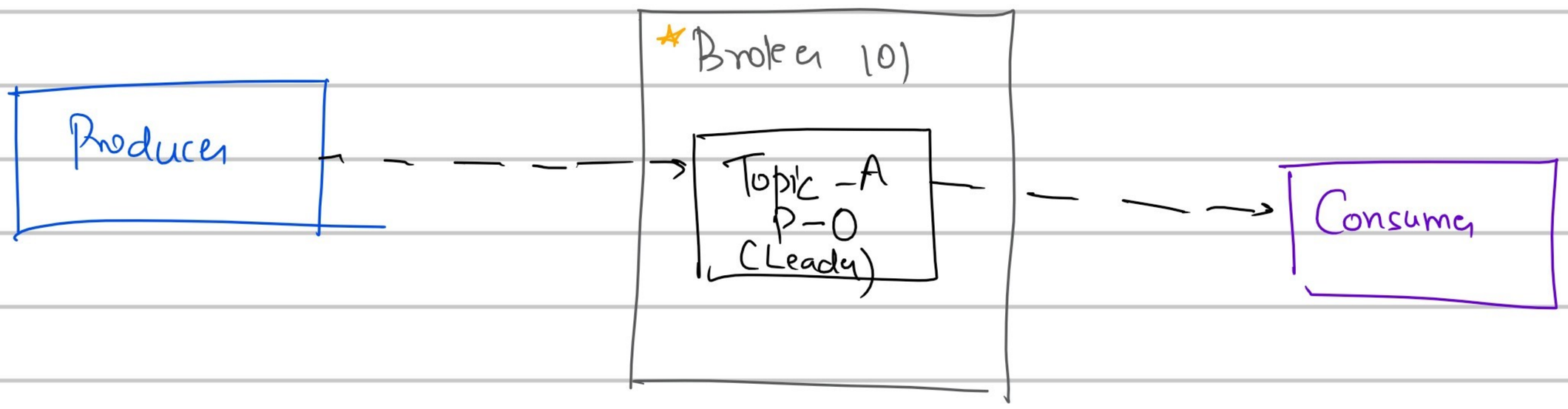
```java
while(true){
    batch+=consumer.poll(Duration.ofMillis(100));
    if isReady(batch){
        doSomethingSynchronous(batch);
        consumer.commitAsync();
    }

}
```

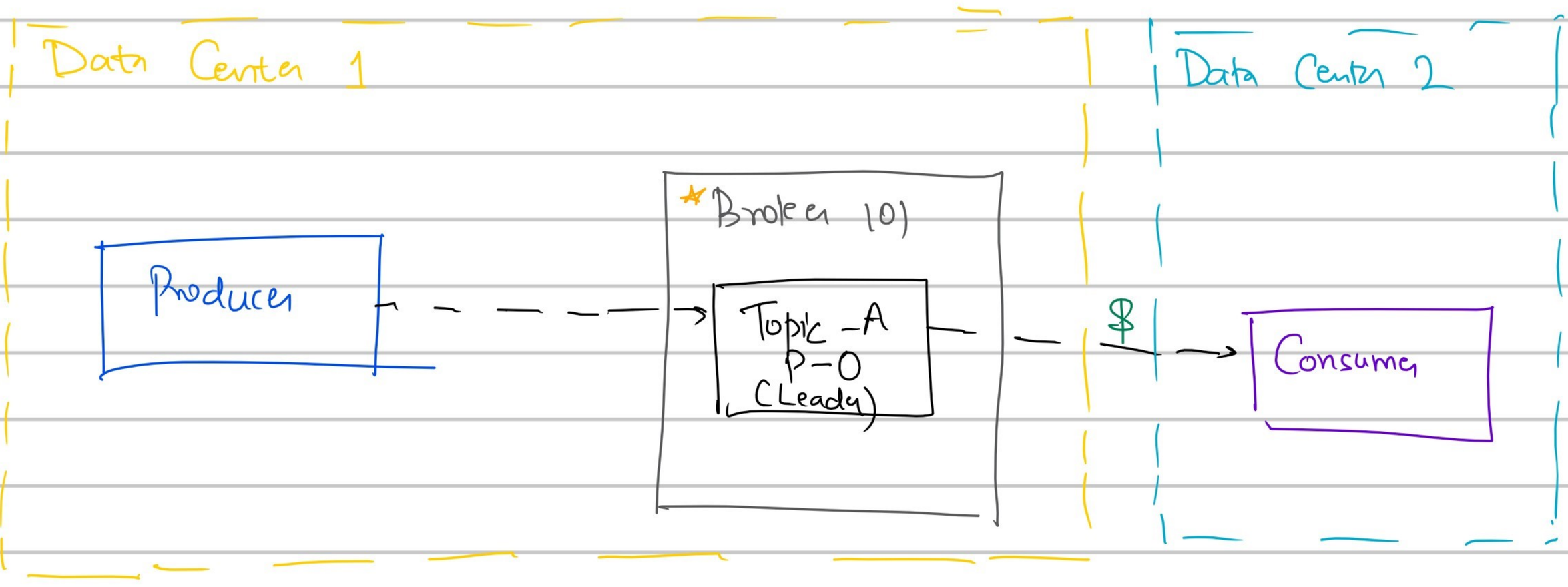
enable.auto.commit = false & synchronous processing of batches.

- You control when you commit offsets & what's the condition for committing them

Example → accumulating records into a buffer & then flushing the buffer to a db & committing the offset asynchronously then

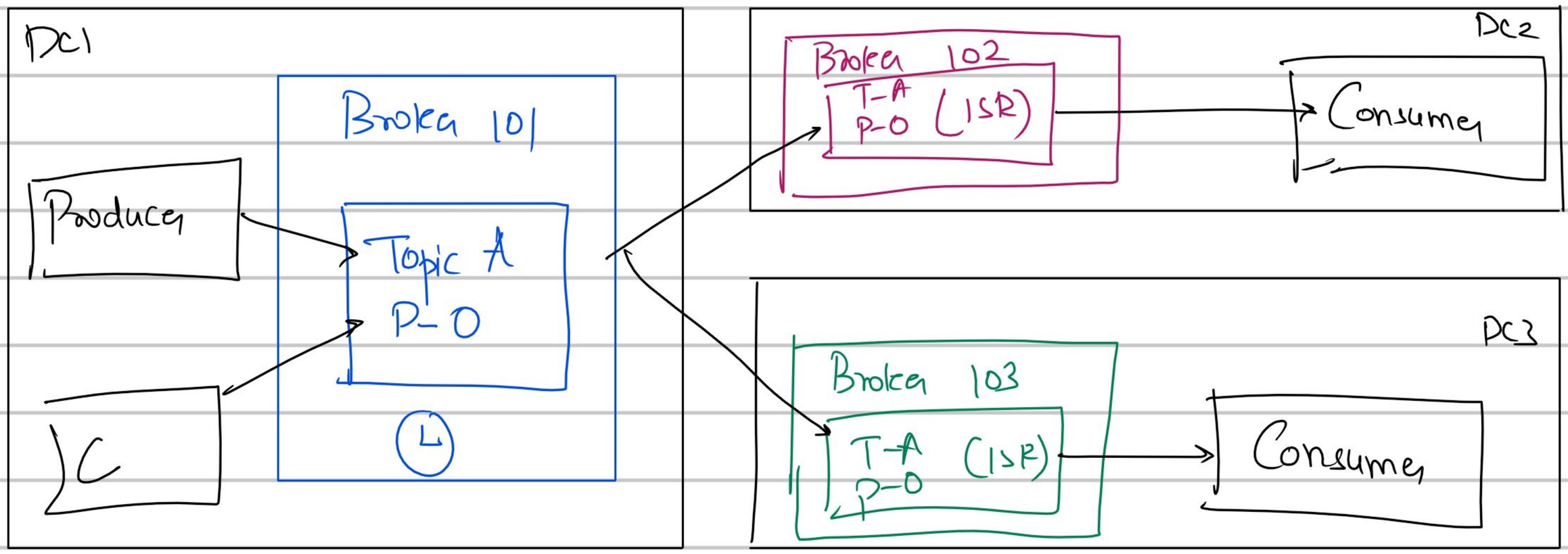
Default Consumer Behavior with Partition Leaders





Rack Awareness

Since Kafka 2.4 it is possible to configure consumers to read from the closest replica



Broker Settings

kafka -2.4 +

rack.id config must be set to data center id (ex: AZ ID in AWS)

Example

rack.id = usw2-az1

- replica.selector.class → org.apache.kafka.common.replica.RackAwareReplicaSelector

Consumer →

client.rack → data center ID ~~the~~ ~~of the~~ where consumer is launched.