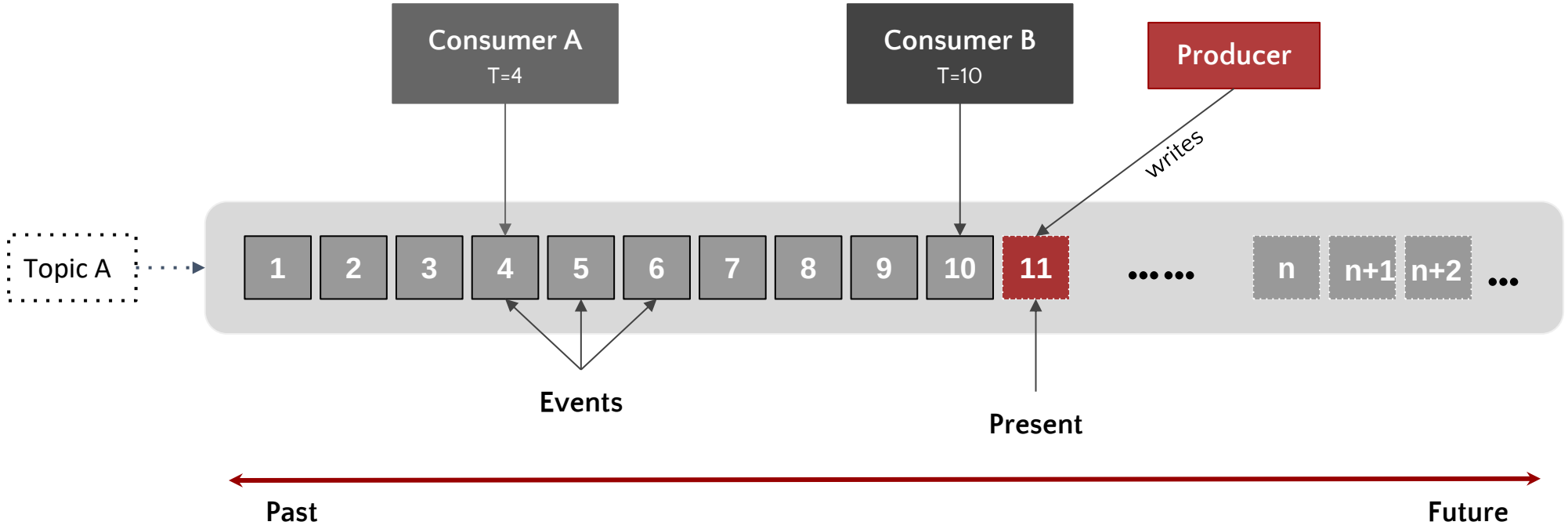

DataCouch
Pioneers in Bleeding Edge Technologies
APACHE KAFKA FOR DEVELOPERS DAY-1

AGENDA

- Working of Kafka
- Delving deeper into Producer Side
- Delving deeper into Consumer Side

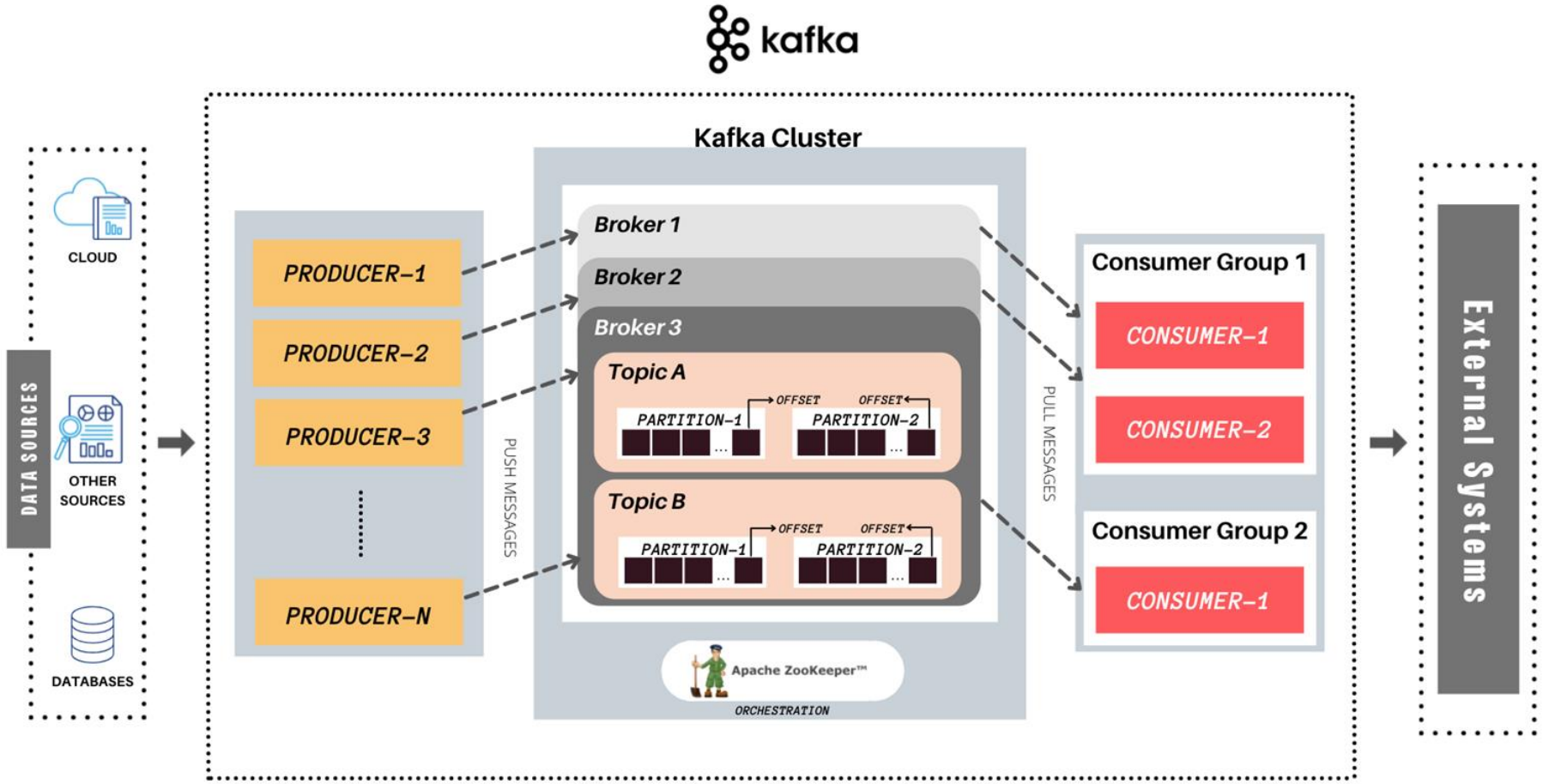
Working of Kafka

Distributed Logs



Working of Kafka

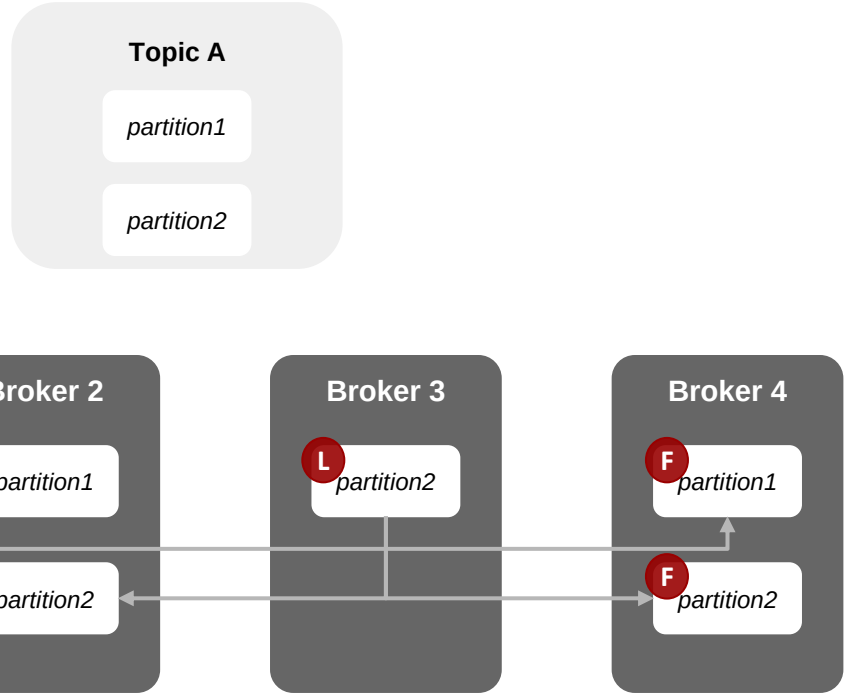
Exploring Kafka Architecture



Working of Kafka

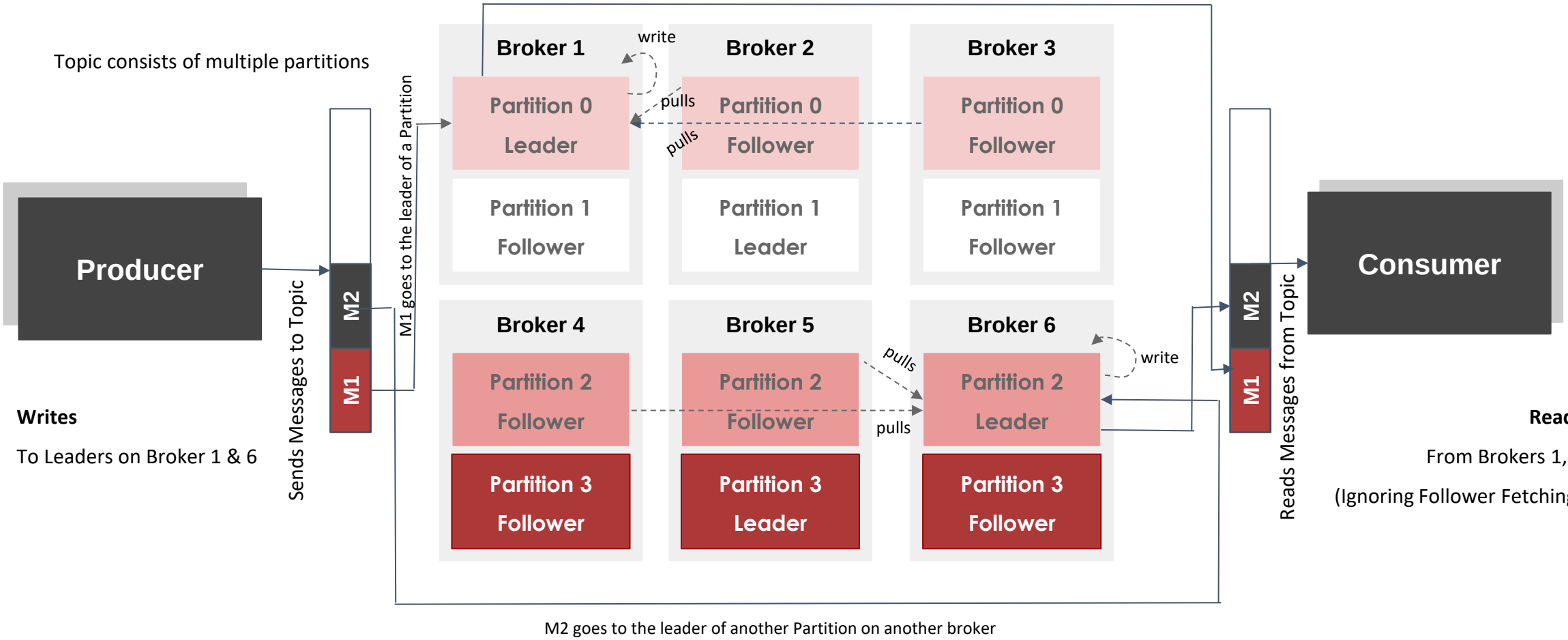
Leader and Follower Partitions

- Kafka topic (Topic A) can have multiple partitions (Partition 1, Partition 2)
- Partitions are stored on various brokers (Broker 1,2,3,4) for enabling parallel processing
- Partitions are replicated over various brokers for ensuring the high availability of data



Working of Kafka

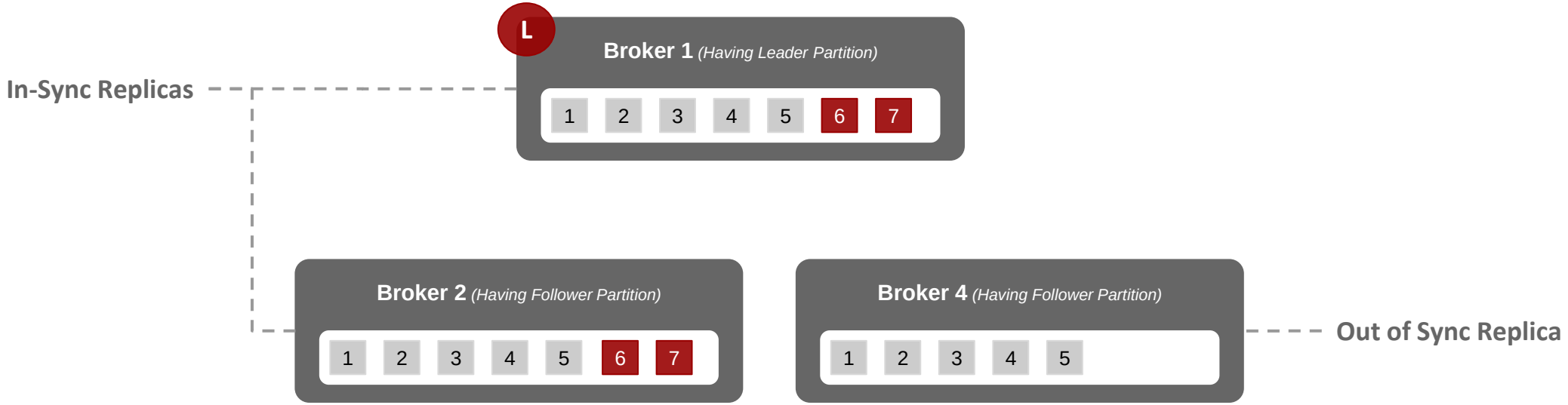
Understanding Replication Mechanism



Working of Kafka

ISR - In-Sync Replica

- An In-Sync Replica (ISR) is a partition on a broker that has the latest data for a given partition. A leader is always an in-sync replica
- A Follower is an In-Sync Replica only if it has fully caught up to the partition it's following (within ***replica.lag.time.max.ms***)
- If a follower broker falls behind the latest data for a partition, we no longer count it as an In-Sync Replica



Working of Kafka

Understanding Replication Mechanism

- **Replication** means having multiple copies of the data, distributed across multiple machines
- Enables us to achieve high availability in case one of the brokers goes down and is unavailable to serve the requests
- Kafka Replication is allowed at the partition level, copies of a partition are maintained at multiple broker instances using the partition's **Write-Ahead Log**
- Amongst all the replicas of a partition, Kafka designates one of them as the "**Leader**" partition and all other partitions are followers
- The **Leader** receives for the **partitions**
- **Zookeeper** takes care of the **synchronization** between the distributed clusters and manages the configurations, controlling and naming
- Each Replica or Node, sends a "**Keep-Alive**" (aka **Heartbeat**) message to Zookeeper at regular intervals

Working of Kafka

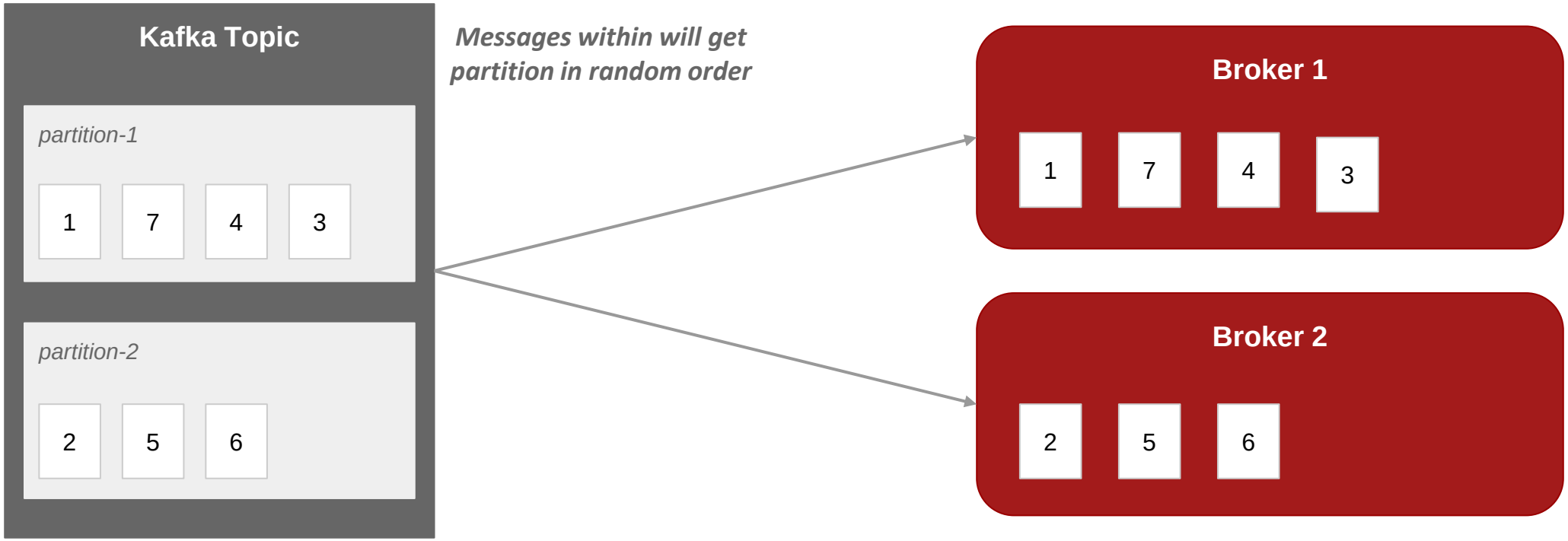
Role of Keyed Messages

- If ordering of messages (w.r.t. a business key in a Topic Partition) matters for a solution then keyed messages are important
- Let’s say you have a Kafka Topic where you want to send order status
- If we receive several status updates about the same order - like “prepared”, “shipped”, and “delivered”, we want to make sure that the applications consumes these statuses in the right order
- In Kafka, the messages are guaranteed to be processed in order only if they share the same key

Working of Kafka

Role of Keyed Messages

Kafka Without Keyed Messages

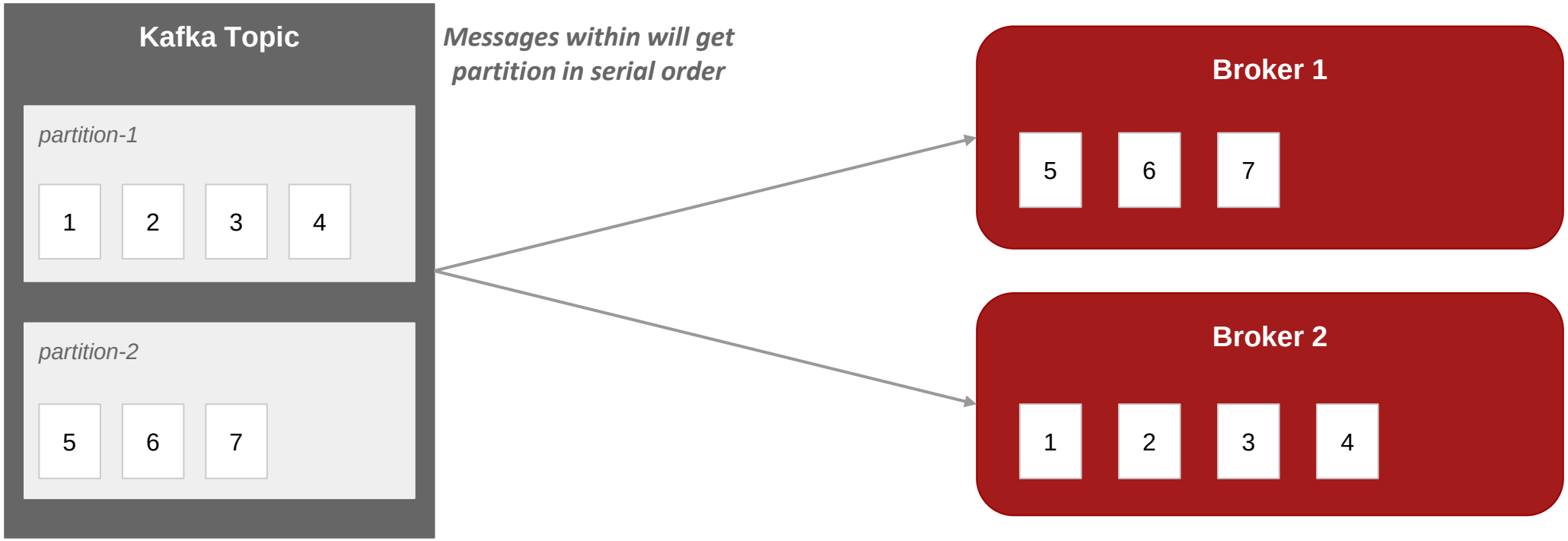


Messages within partitions are read by brokers in random order…

Working of Kafka

Role of Keyed Messages

Kafka with Keyed Messages



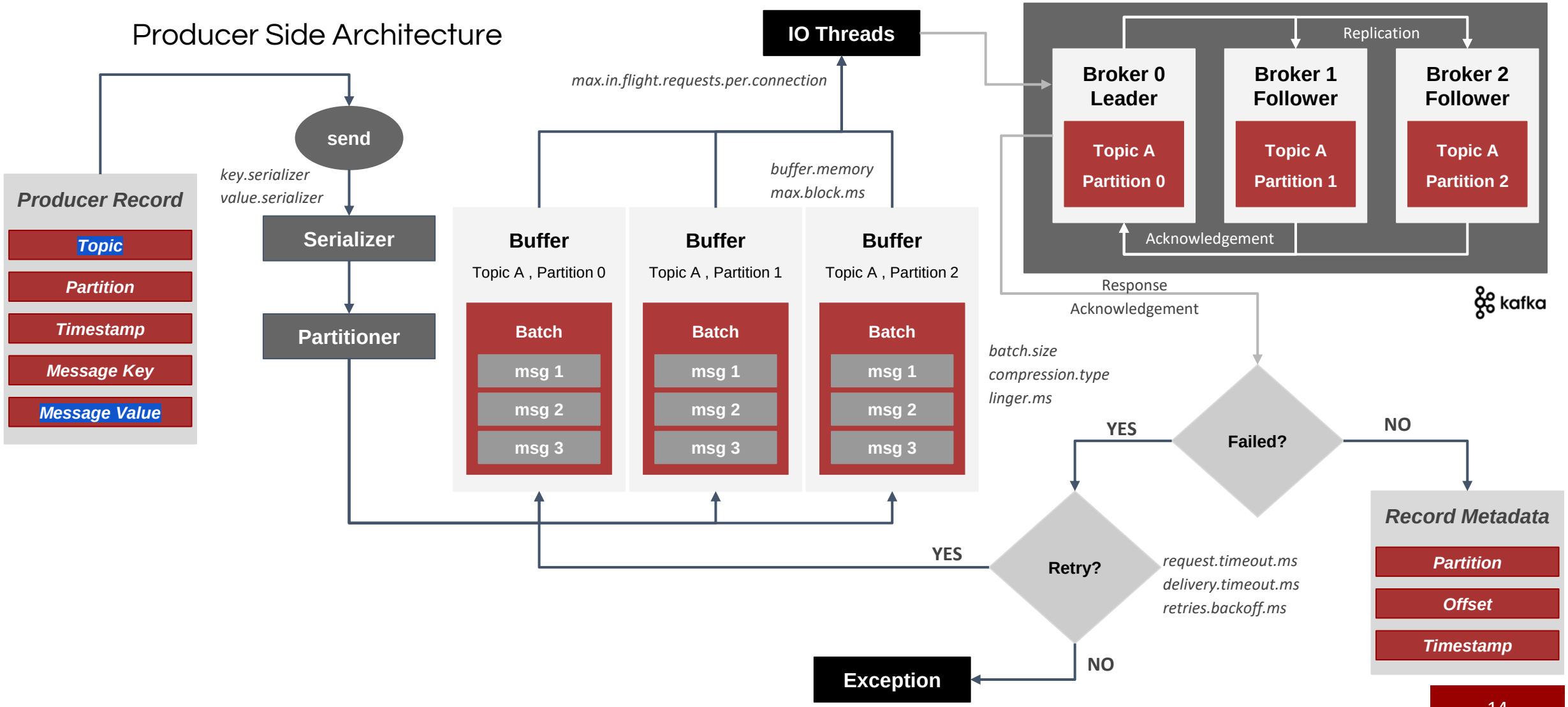
Messages within partitions are read by brokers in an ordered way only…

AGENDA

- Working of Kafka
- **Delving deeper into Producer Side**
- Delving deeper into Consumer Side

DELVING DEEPER INTO **PRODUCER SIDE**

Producer Side Architecture



DELVING DEEPER INTO **PRODUCER SIDE**

Notes on Architecture

- **Send()**- Send doesn't send data to Kafka Brokers directly. It adds the record to a buffer of pending records to be sent and returns immediately. This buffer of pending records helps to batch together individual records for efficiency
- **Serializer**- It helps serialize the key and the value objects that the user provides with their ProducerRecord into bytes
- Producer can send data to a specific Topic Partition but it is hardcoding approach (not recommended)
- **Partitioner**- Partition assignment to every ProducerRecord is done by the Partitioner. There are two types of partitioning strategies -
  - **Round Robin Partitioning** - This is used when the key is Null in ProducerRecord. Partitions are assigned in Round Robin fashion
  - **Hash Key-based Partitioning** - This is used when ProducerRecord has a key. Partitions are determined based on the hash of the key for a particular message

DELVING DEEPER INTO **PRODUCER SIDE**

Notes on Architecture

- **Buffer** - Enables us to store data locally at the producer side for improving its performance
- **I/O Threads** - Threads responsible for sending batches of records to the Brokers

DELVING DEEPER INTO **PRODUCER SIDE**

Producer Side Configurations

Let's e er by explori erties:

bootstrap.servers

*List of **host:port** pairs of brokers that the producer will use to establish the initial connection to the Kafka cluster*

key.serializer

*Name of a class that will be used to serialize the keys of the records, **key.serializer** should be set to the name of a class that implements the **org.apache.kafka.common.serialization.Serializer** interface*

value.serializer

*The name of a class that will be used to serialize the values of the records we will produce to Kafka, set **value.serializer** to a class that will serialize the message value object*

DELVING DEEPER INTO **PRODUCER SIDE**

Producer API w.r.t. Java

```
private Properties kafkaProps = new Properties();
kafkaProps.put("bootstrap.servers" ,  "broker1:9092,broker2:9092");
kafkaProps.put("key.serializer" , "org.apache.kafka.common.serialization.StringSerializer");
kafkaProps.put("value.serializer" , "org.apache.kafka.common.serialization.StringSerializer");
producer = new KafkaProducer<String, String>(kafkaProps);
```

- Initializing Properties Object
- Using StringSerializer for messages having String based Keys and Values
- Instantiating a new Producer by configuring the appropriate key and value types and passing the Properties object

DELVING DEEPER INTO **PRODUCER SIDE**

How data distributed across the cluster?

How default partitioners works:

- If a partition is specified in the record then send data to that partition (implicit buffering is happening)
- If no partition is specified but a key is present, choose a partition based on a hash of the key - It can be used when we want to distribute the data based on a key
- The following formula is used to determine the partition:

  hashCode(key) % NoOfPartitions
- If no partition or key is present, choose a partition in a round-robin fashion - If we are not bothered about which partition our data is going to, this strategy can be used

DELVING DEEPER INTO **PRODUCER SIDE**

Case for Custom Partitioner

- Default strategies may work well in the beginning
- Enables us to overcome Data Skew Issues
- Custom Partitioner provides us an ability to have flexibility and control over the distribution of data
- Helps us to send data of a particular business key e.g. Customer Id to the same partition even when the key is composite for instance - Customer Id and Date
- Although the customer ID is fixed, the date can change, and we will end up with a different hash code, which will mean that the data for the same customer will go to different partitions

DELVING DEEPER INTO **PRODUCER SIDE**

Understanding Delivery Semantics of Apache Kafka - At-Most Once

Apache Kafka supports 3 message delivery semantics:

- ***at-most-once***
- ***at-least-once***
- ***exactly-once***

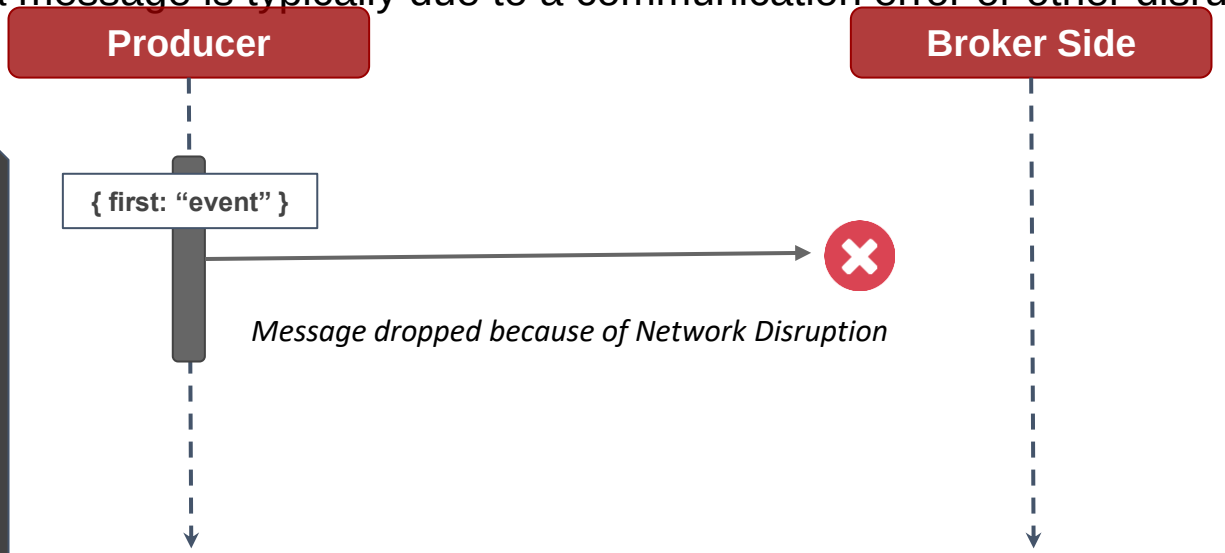
Let's explore each one of them one-by-one…

DELVING DEEPER INTO **PRODUCER SIDE**

Understanding Delivery Semantics of Apache Kafka - At-Most Once

At-Most-Once Delivery Semantic

- ❑ Message is delivered either one time only or not at all
- ❑ If there is a failure to deliver a message then there is loss of Data
- ❑ Failure to deliver a message is typically due to a communication error or other disruption



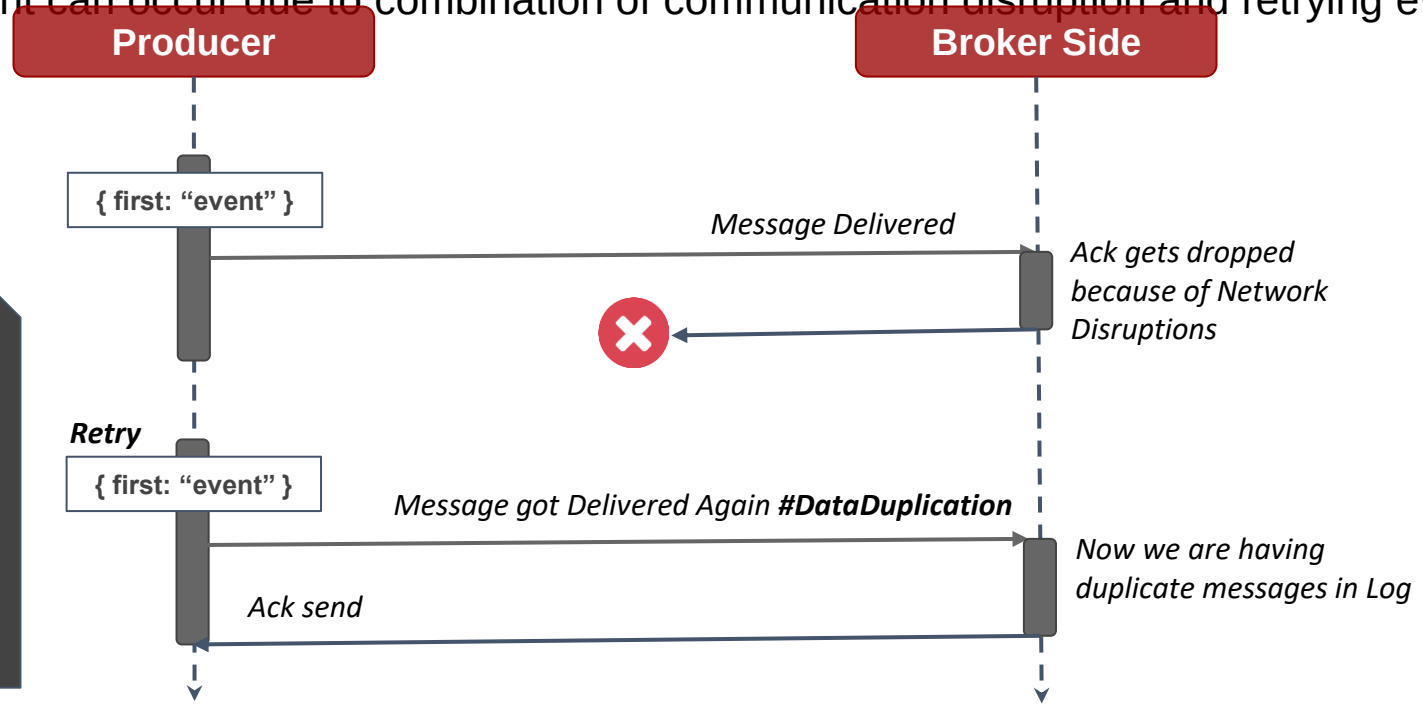
Note: Ideal for applications where the guaranteed delivery doesn't impact the business e.g. Twitter sentiment analysis - even if some messages are not delivered out of millions of messages then also it would not impact the general sentiment

DELVING DEEPER INTO **PRODUCER SIDE**

Understanding Delivery Semantics of Apache Kafka - At-Least Once

At-Least-Once Delivery Semantic

- ❑ Message is delivered once or many times, but will never be lost
- ❑ Duplication of event can occur due to combination of communication disruption and retrying events



Note: Ideal for applications where receiving every message is important e.g. financial transaction messages. In case of duplicate messages, the logic can be applied to de-duplicate data

DELVING DEEPER INTO **PRODUCER SIDE**

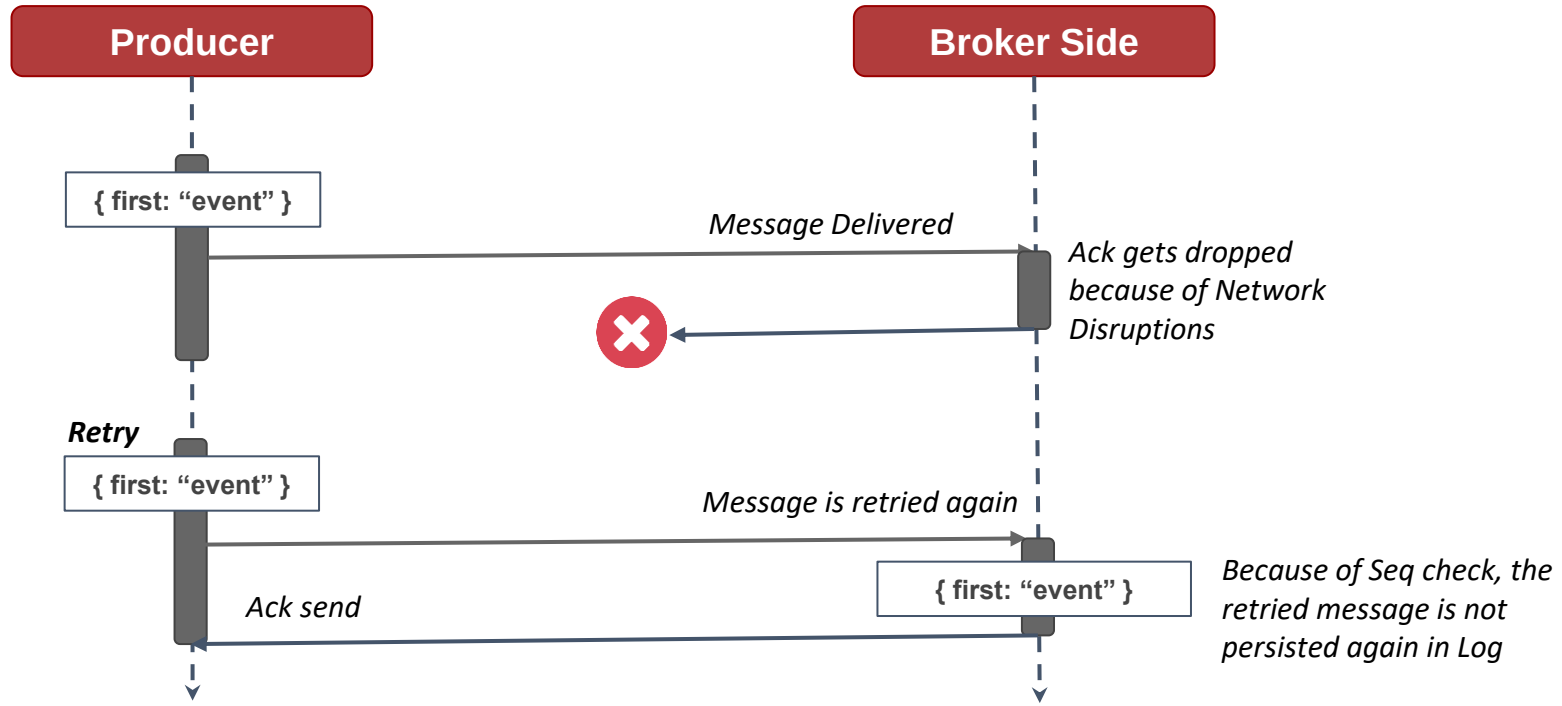
Understanding Delivery Semantics of Apache Kafka - Exactly Once

Exactly Once Delivery Semantic

- ❑ Message is delivered only one time
- ❑ Message can neither be dropped nor be duplicated
- ❑ Achieved by making changes in the configuration of Producer side and Consumer side along with use of Transactional APIs

DELVING DEEPER INTO **PRODUCER SIDE**

Understanding Delivery Semantics of Apache Kafka - Exactly Once



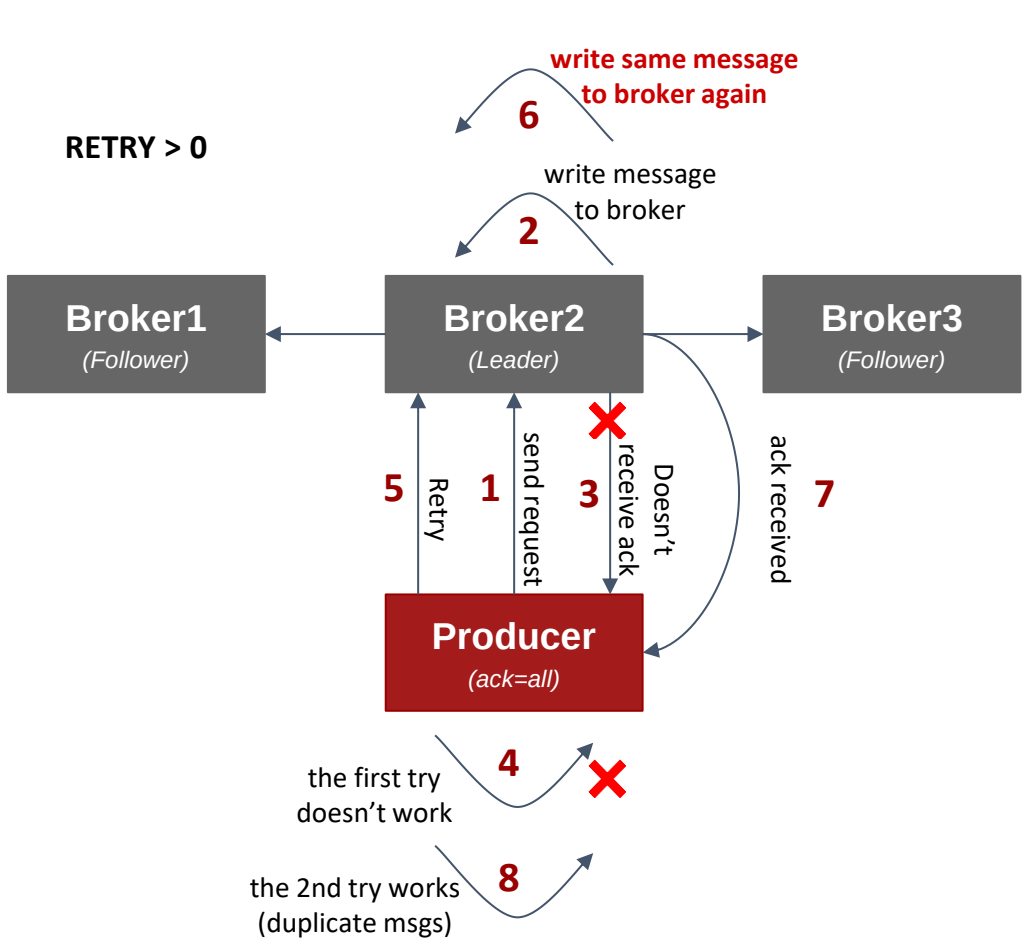
DELVING DEEPER INTO **PRODUCER SIDE**

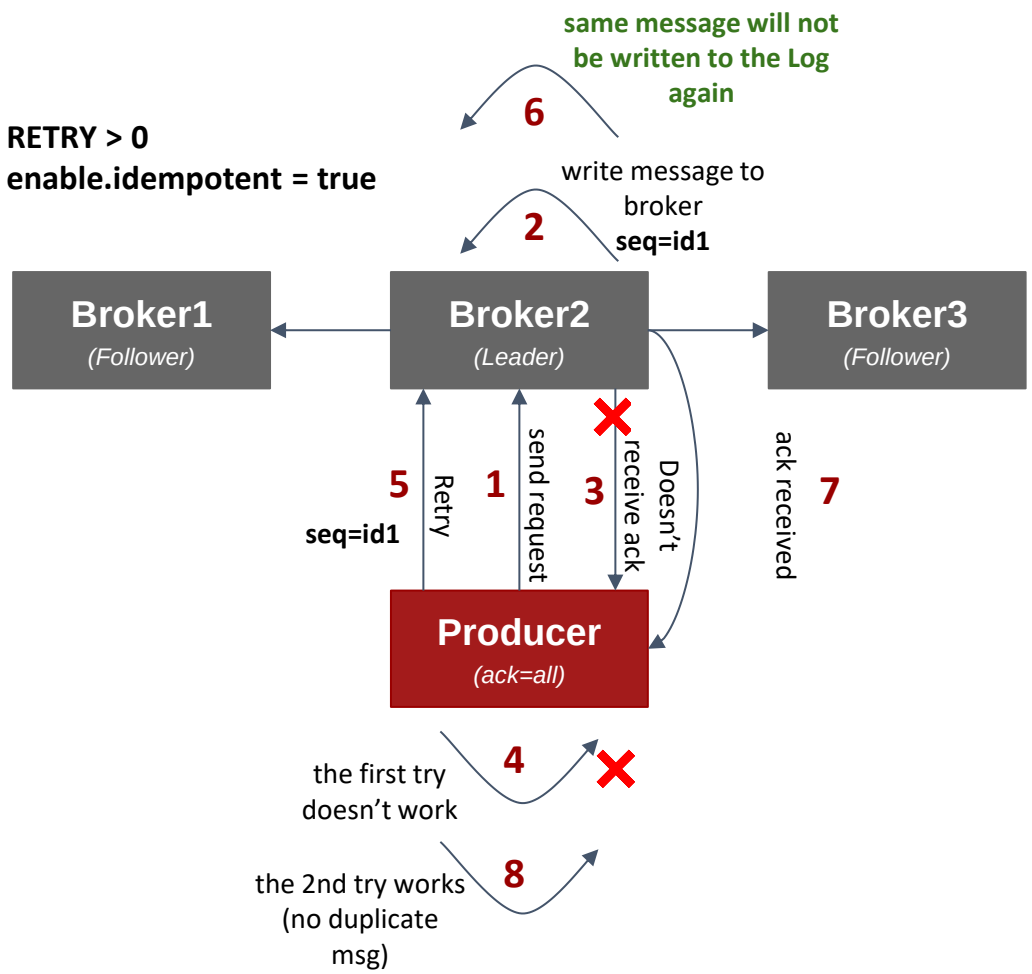
Idempotent Producer

- Idempotence is the property of certain operations to be applied multiple times without changing the result
- When turned on, a producer will make sure that just one copy of a record is being published to the stream
- The default value is false (< Kafka 3.x), meaning a producer may write duplicate copies of a message to the stream
- To turn idempotence on, configure producer side property to ***enable.idempotence=True***

DELVING DEEPER INTO **PRODUCER SIDE**

Idempotent Producer





DELVING DEEPER INTO **PRODUCER SIDE**

Recommended Configurations

- ***enable.idempotence*** - determines whether the producer may write duplicates of a retried message to the topic partition
- ***acks*** - must be set to **all**
- ***min.insync.replicas*** - minimum required number of in-sync replica partitions have acknowledged receipt of the message before itself acknowledging the message. It should be set to at-least 2
- ***retries*** - Must be non-zero. If it is set to zero then the Producer will not re-attempt to send the message

AGENDA

- Working of Kafka
- Delving deeper into Producer Side
- Delving deeper into Consumer Side

DELVING DEEPER INTO **CONSUMER SIDE**

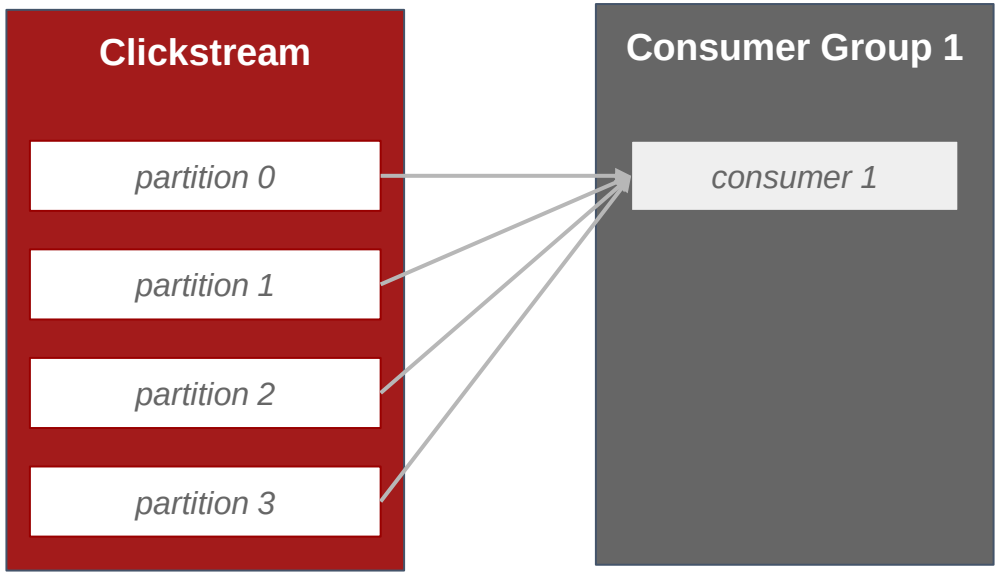
Role of Consumer Group

- Consumer Group helps in providing scalability at the Consumer side
- Consumers are typically part of a Consumer Group
- When multiple consumers are subscribed to a topic and belong to the same consumer group, each consumer in the group will receive messages from a different subset of the partitions in the topic

DELVING DEEPER INTO **CONSUMER SIDE**

Partitions Assignment to a single Consumer

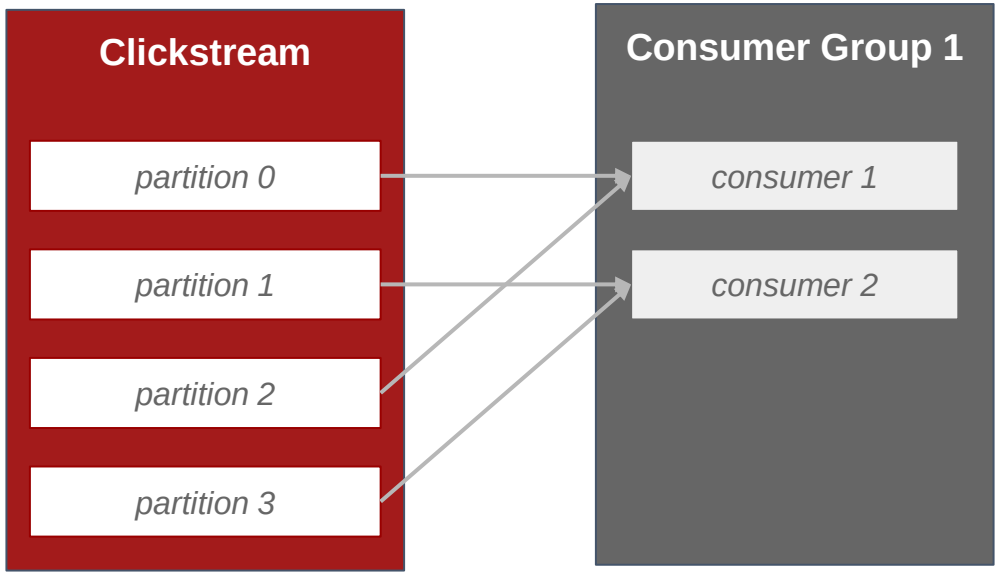
- Let's take topic ***clickstream*** with four partitions. Consumer 1 (only consumer in CG1) subscribes to this topic then all messages from all the four partitions go to this Consumer 1



DELVING DEEPER INTO **CONSUMER SIDE**

Partitions Assignment to multiple Consumers in a Consumer Group

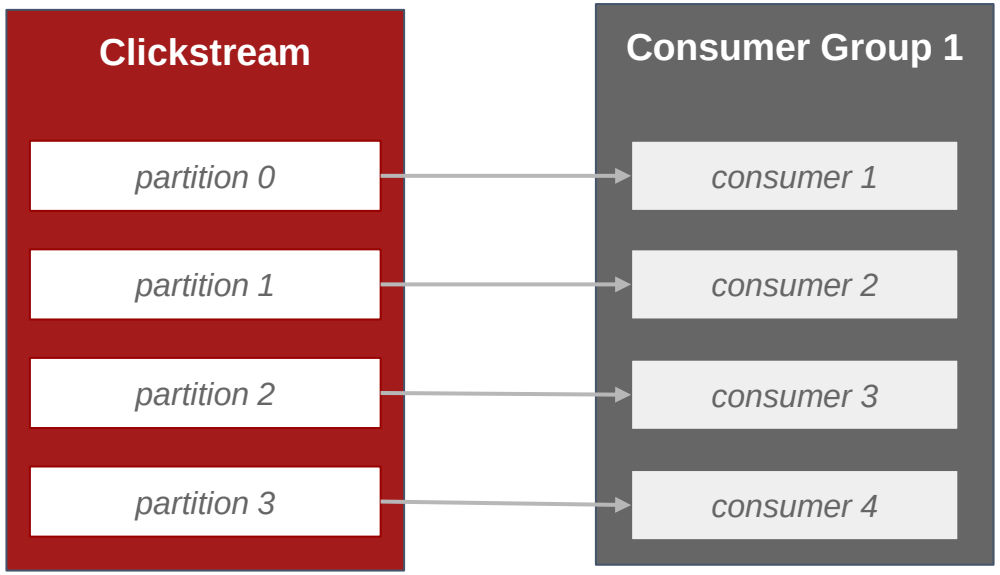
- Let's add another Consumer C2 to group CG1 then each consumer will only get messages from two partitions. May be messages from partition 0 and 2 go to C1 and messages from partitions 1 and 3 go to consumer C2



DELVING DEEPER INTO **CONSUMER SIDE**

Partitions Assignment to multiple Consumers in a Consumer Group

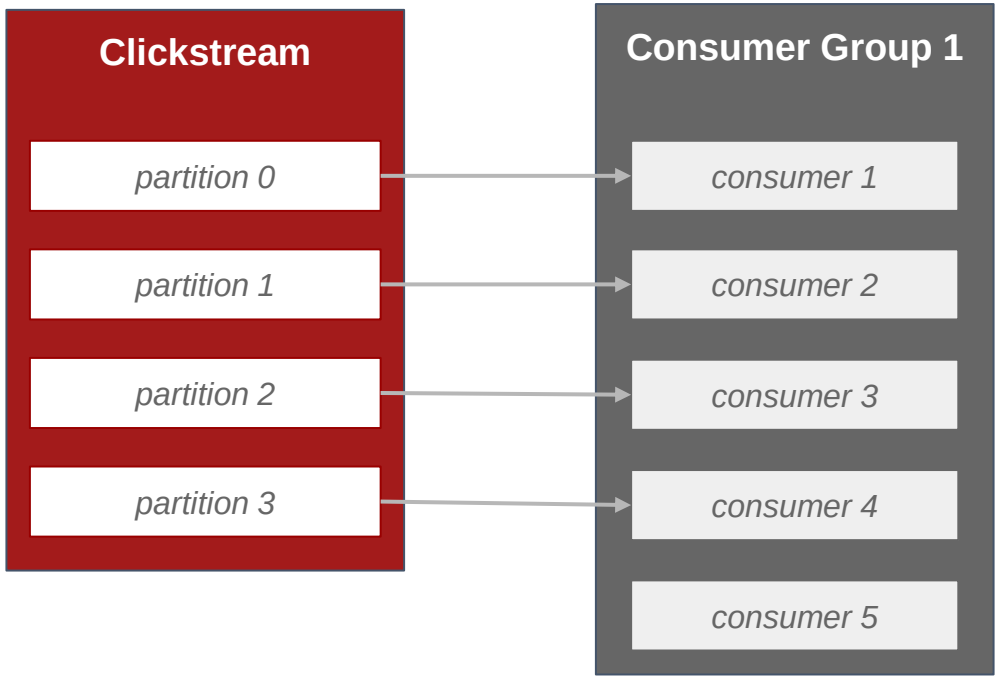
- If CG1 has four consumers, then each will read messages from a single partition



DELVING DEEPER INTO **CONSUMER SIDE**

Partitions Assignment to multiple Consumers in a Consumer Group

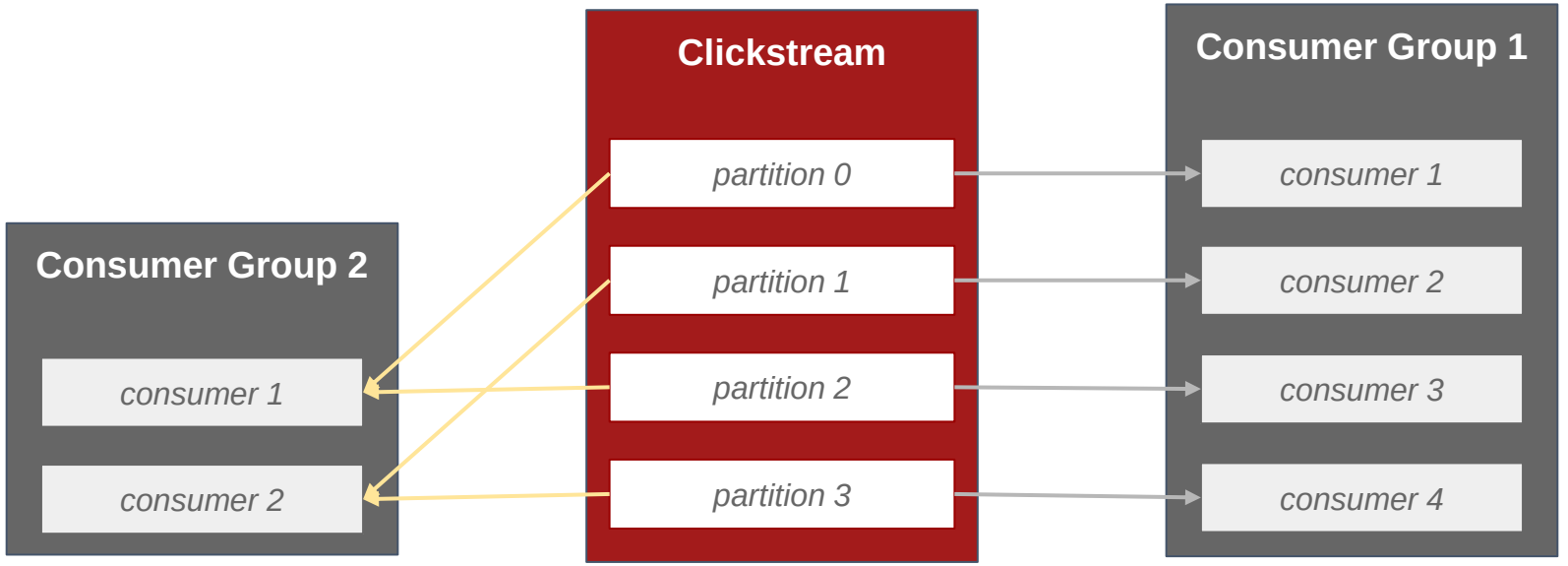
- If we add more consumers to a single group with a single topic than we have partitions, some of the consumers will be idle and get no messages at all



DELVING DEEPER INTO **CONSUMER SIDE**

Partitions Assignment to multiple Consumers in multiple Consumer Groups

- In the previous example, if we add a new consumer group CG2 with a single consumer, this consumer will get all the messages in topic T1 independent of what CG1 is doing
- CG2 can have more than a single consumer, in which case they will each get a subset of partitions, just like we showed for CG1, but CG2 as a whole will still get all the messages regardless of other consumer groups



DELVING DEEPER INTO **CONSUMER SIDE**

Consumer Offsets

- The consumer offset is a way of tracking the sequential order in which messages are received by Kafka topics
- It allows you to keep track of how far away any given consumer is from being able to read data from a topic
- This is important for nearly all Kafka use cases and can be an absolute necessity in certain instances, such as financial services
- Whenever we try to read data from topic, it returns records written to Kafka that consumers in our group have not read yet
- This means that we have a way of tracking which records were read by a consumer of the group

DELVING DEEPER INTO **CONSUMER SIDE**

Consumer Rebalance Process

- A rebalance is a process that moves partition ownership from one consumer to another
- It provides high availability and scalability, but in the normal course of events it is fairly undesirable since consumers can’t consume messages during a rebalance and so are unavailable for a short period of time
- If a consumer has any cached data, it will need to refresh its caches—slowing down the application until the consumer sets up its state again

DELVING DEEPER INTO **CONSUMER SIDE**

Consumer Rebalance Process

- Consumers maintain membership in a consumer group and ownership of the partitions assigned to them by sending heartbeats to a Kafka broker designated as the group coordinator
- This keeps consumers alive and well, so long as they are sending heartbeats at regular intervals. Heartbeats are sent when a consumer polls, after it has consumed records from its partitions, and when it commits batches of records to partitions it has buffered
- If a consumer does not send any heartbeats for long enough, its session will time out and cause rebalance to occur

DELVING DEEPER INTO **CONSUMER SIDE**

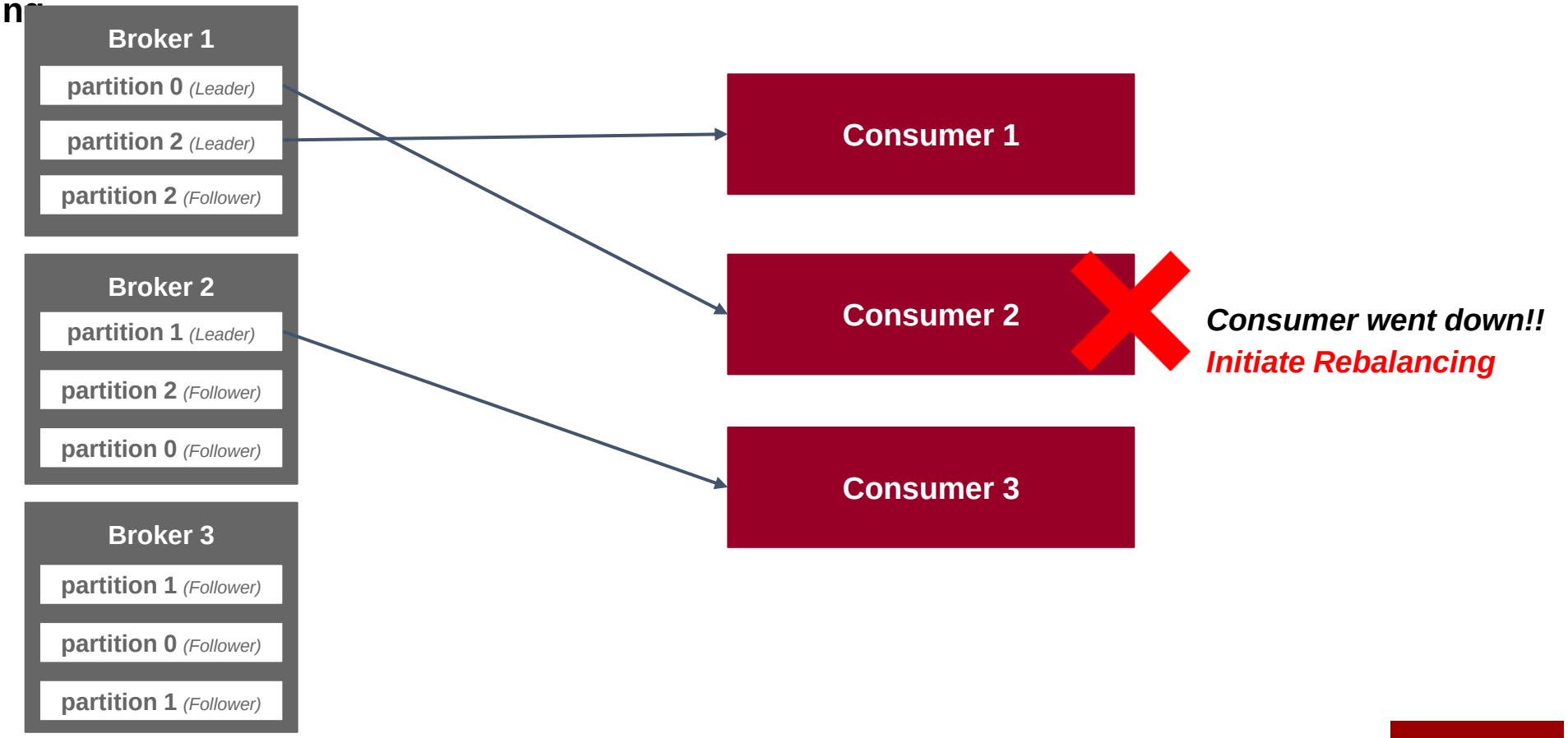
Consumer Rebalance Process

- If a consumer crashes and stops processing messages, the group coordinator will wait for a few seconds without receiving heartbeats before deciding that the consumer is dead
- At this point, no messages will be processed from that consumer's partitions
- When closing a consumer cleanly, the consumer notifies its group coordinator that it is leaving, and the group coordinator triggers a rebalance immediately at this time—reducing the gap in processing time

DELVING DEEPER INTO **CONSUMER SIDE**

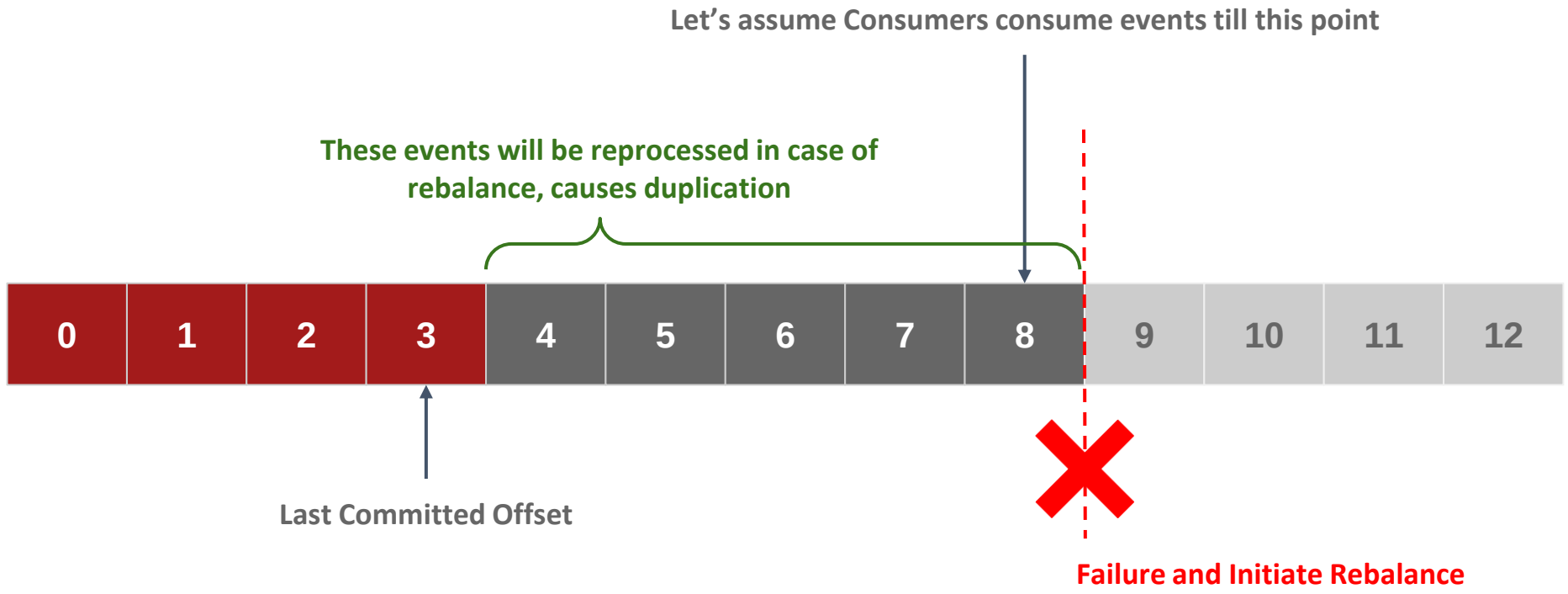
Consumer Rebalance Process

Before Rebalancing



DELVING DEEPER INTO **CONSUMER SIDE**

Consumer Rebalance Process



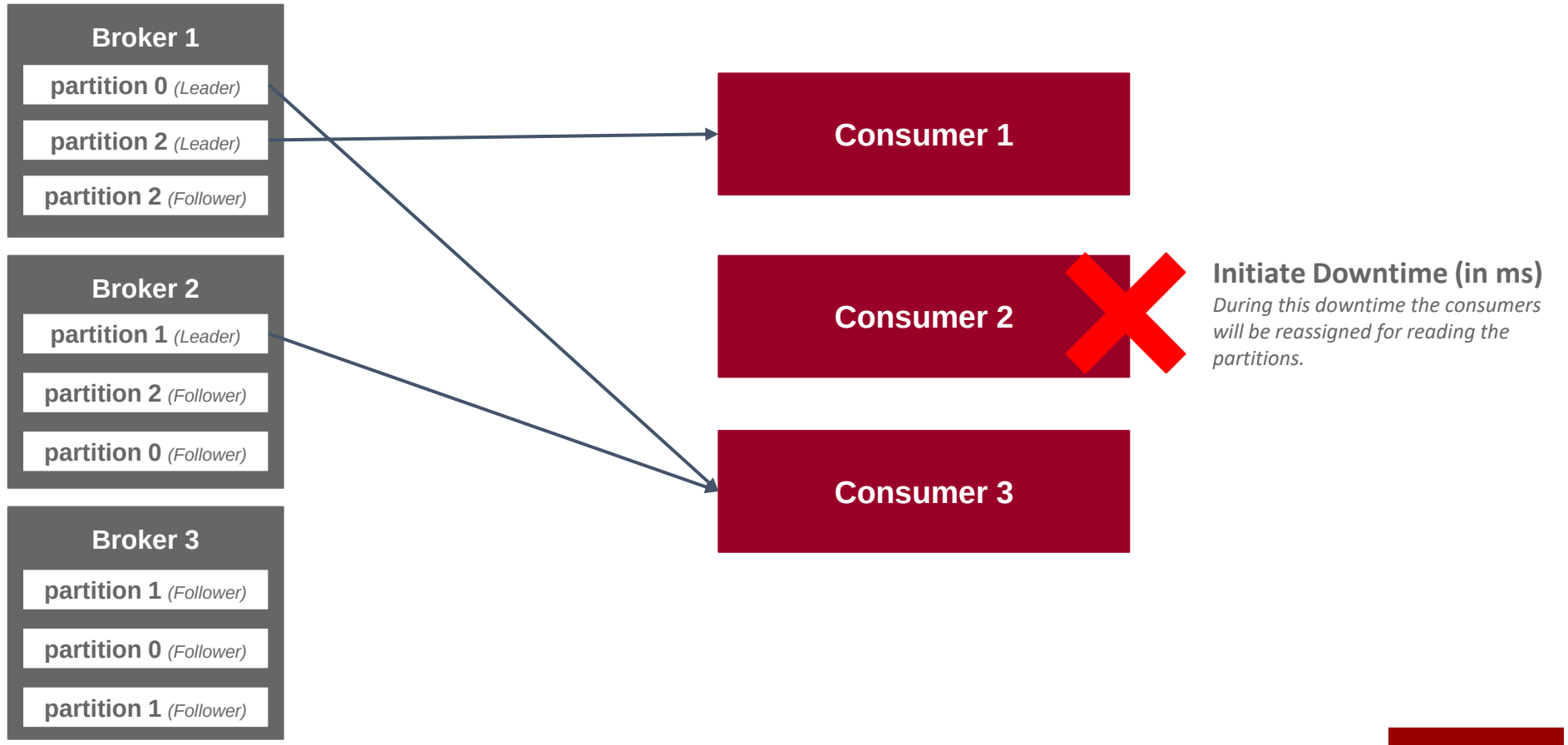
Initiate Downtime (in ms)

During this downtime the consumers will be reassigned for reading the partitions.

DELVING DEEPER INTO **CONSUMER SIDE**

Consumer Rebalance Process

After Rebalancing



DELVING DEEPER INTO **CONSUMER SIDE**

Kafka Consumer API w.r.t. Java

- To log records in Kafka, you first create a **KafkaConsumer** instance
- The process of creating a consumer is similar to that of creating a producer—you create a Java Properties instance with the properties you want to pass to the consumer
- We will discuss all the properties, but three mandatory properties: **bootstrap.servers, key.deserializer** and **value.deserializer**

DELVING DEEPER INTO **CONSUMER SIDE**

Kafka Consumer API w.r.t. Java

- The first property, **bootstrap.servers**, is the connection string to a Kafka cluster
- It is used the same way as in **KafkaProducer**
- The other two properties are key.deserializer and **value.deserializer**, which provide classes that can take a byte array and turn it into a Java object

DELVING DEEPER INTO **CONSUMER SIDE**

Kafka Consumer API w.r.t. Java

The consumer is constructed using a Properties file just like the other Kafka clients. In the example below, we provide the minimal

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "clickstream");
props.put("key.deserializer", StringDeserializer.class.getName());
props.put("value.deserializer", StringDeserializer.class.getName());
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
```

DELVING DEEPER INTO **CONSUMER SIDE**

Kafka Consumer API w.r.t. Java

To begin consumpti

```
consumer.subscribe(Arrays.asList("clickstream1", "clickstream2"));
```

from

- The consumer can coordinate with any other consumers in the same group to get a partition assignment
- This is all handled automatically when you begin consuming data from a topic, but later, we will show how you can assign partitions manually using the assign API
- The subscribe method is not incremental: you must include the full list of topics that you want to consume from

DELVING DEEPER INTO **CONSUMER SIDE**

Kafka Consumer API - Poll()

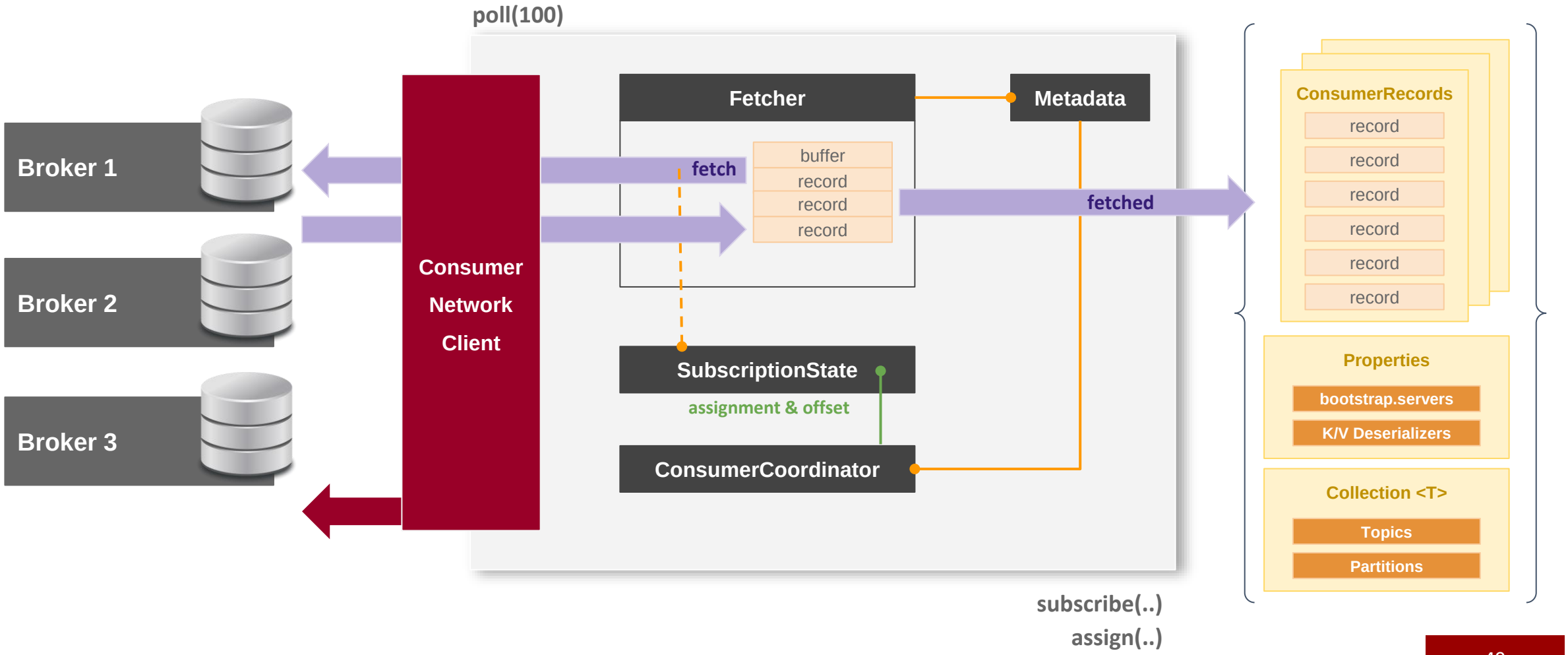
If, consumer needs to be able to fetch data in parallel, potentially from many partitions for many topics likely spread across many b

```
try {
  while (running) {
    ConsumerRecords<String, String> records = consumer.poll(1000);
    for (ConsumerRecord<String, String> record : records)
      System.out.println(record.offset() + ": " + record.value());}}
finally
```

After subscribing to a topic, start the event loop to get a partition assignment and begin fetching data. All we need to do is call poll in a loop and the consumer handles the rest

DELVING DEEPER INTO **CONSUMER SIDE**

Kafka Consumer API - Poll()



DELVING DEEPER INTO **CONSUMER SIDE**

Key Consumer Side Configurations

Property	Description
fetch.min.bytes	Allows a consumer to specify the minimum amount of data that it wants to receive from the broker when fetching records
fetch.max.wait.ms	Tells Kafka how long to wait until it has enough data to send before responding to the consumer
max.partition.fetch.bytes	Controls the maximum number of bytes the server will return per partition
session.timeout.ms	Amount of time a consumer can be out of contact with the brokers while still considered alive defaults to 10 seconds

DELVING DEEPER INTO **CONSUMER SIDE**

Key Consumer Side Configurations

Property	Description
auto.offset.reset	Controls the behavior of the consumer when it starts reading a partition for which it doesn't have a committed offset or if the committed offset it has is invalid
enable.auto.commit	Controls whether the consumer will commit offsets automatically, and defaults to true
max.poll.records	Controls the maximum number of records that a single call to poll() will return

AGENDA

- Working of Kafka
- Delving deeper into Producer Side
- Delving deeper into Consumer Side


THANK YOU