
TEKCRUX
We ReDefine IT
kafka

Module 4: Kafka Internals

Objectives

After completing of this module, you should be able to:

- ✓ Understanding Kafka Internals
- ✓ Explain how replication works in Kafka
- ✓ Difference between In-Sync & Out-of-Sync Replicas
- ✓ Classify and Describe Requests in Kafka
- ✓ Configure Broker, Producer and Consumer
- ✓ Validate System Reliabilities
- ✓ Configure Kafka for Performance Tuning

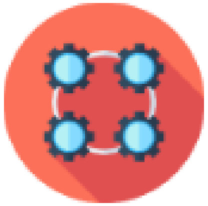
Let’s Learn about Kafka Internals

Kafka Internals

Knowing about Kafka Internals will help you understand how Kafka behaves the way it does

Three core concepts that explains Kafka Internals :

1 How Kafka Replication Works?



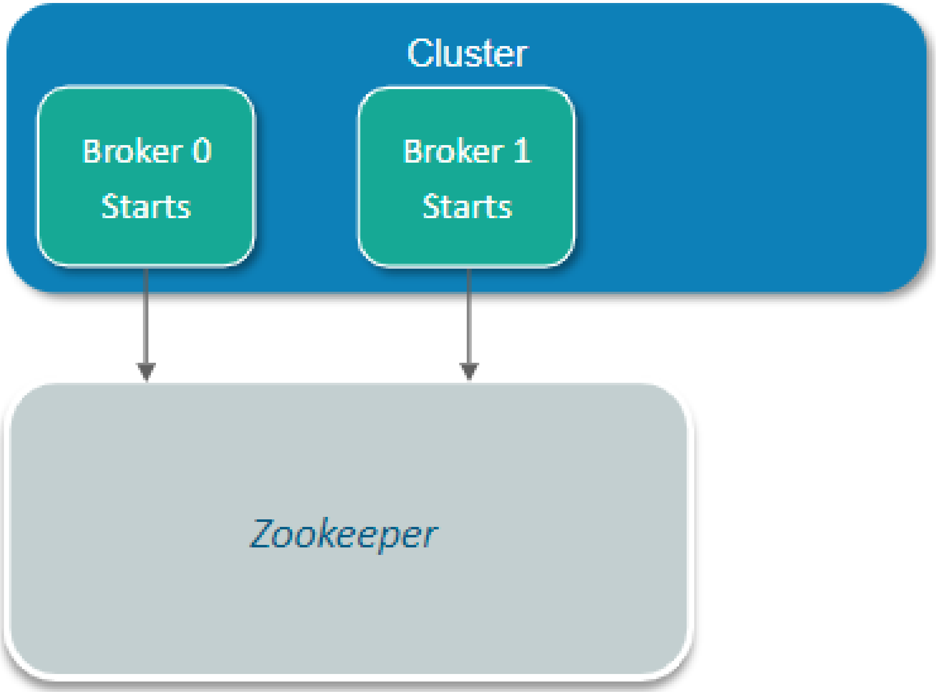
2 How Kafka handles requests from producers and consumers?

3 How Kafka handles storage such as file format and indexes?

Kafka Cluster Membership
&
Controller

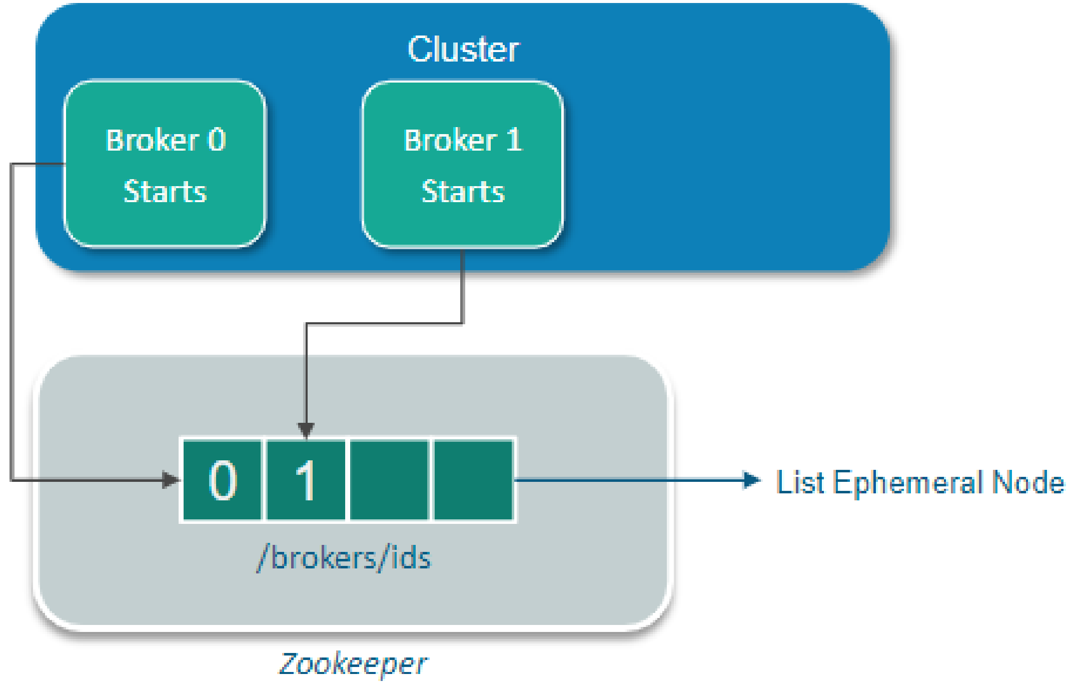
Cluster Membership

- Brokers residing in a Kafka Cluster is considered a member of that cluster
- Kafka uses Apache Zookeeper to maintain the list of brokers that are currently members of a cluster
- Every broker has a unique identifier that is either set in the broker configuration file or automatically generated



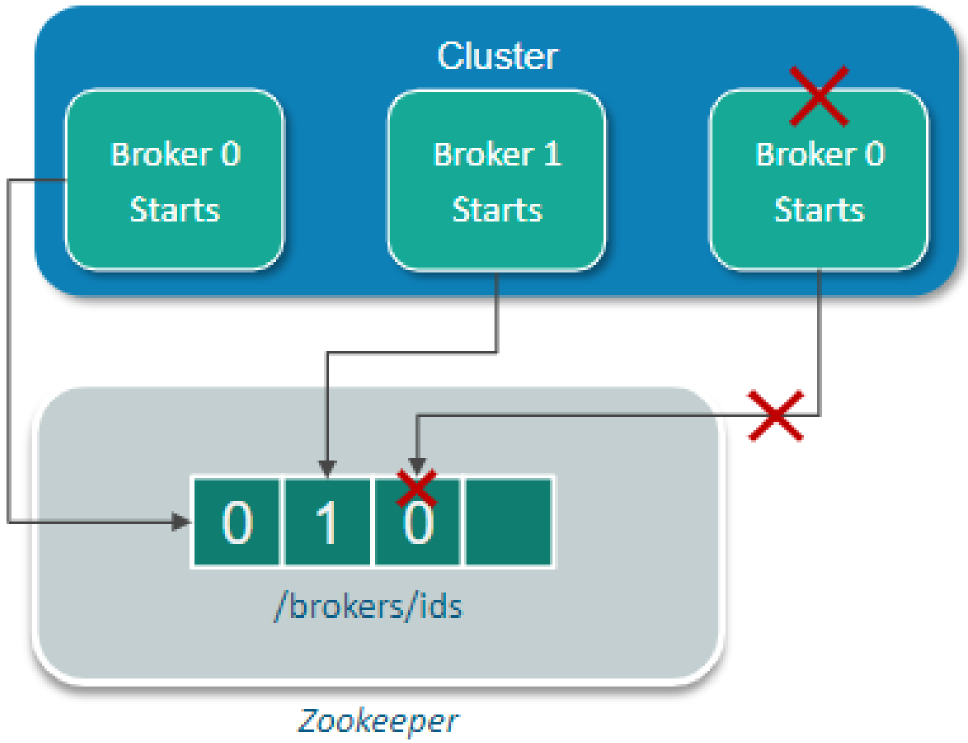
Cluster Membership

- Brokers are registered in Zookeeper's path */brokers/ids,* where Producers and Consumers subscribe to receive notifications regarding the broker
- Every time a broker process starts, it registers itself with its ID in Zookeeper by creating an *ephemeral node*



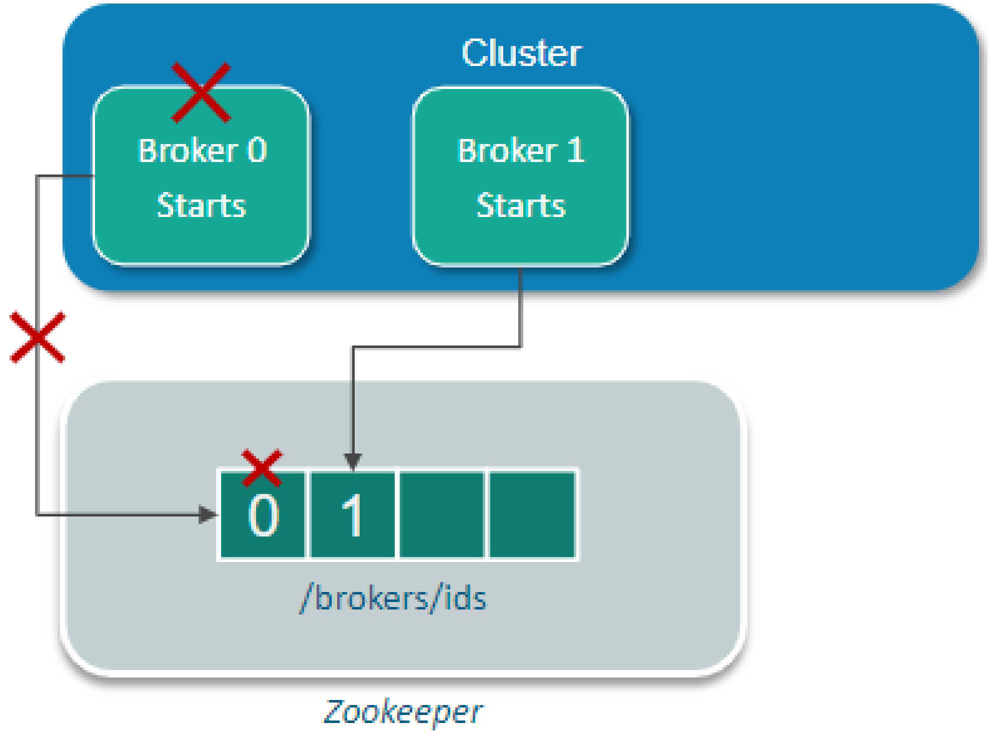
Cluster Membership

- If you try to start another broker with the same ID, you will get an error
- The new broker will try to register, but it will fail



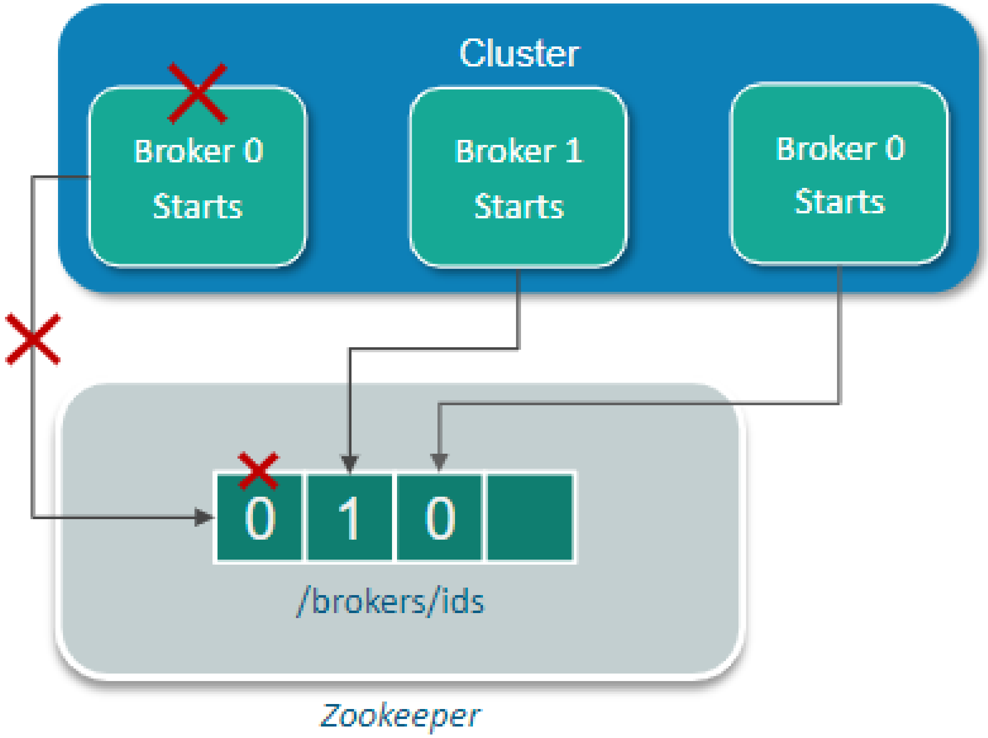
Cluster Membership

- When Kafka broker loses connectivity to ZooKeeper the ephemeral node that created it is also removed
- Even though the broker is gone, the broker ID still exists in some producers and consumers



Cluster Membership

- If we start a new broker with the ID of the old one, it will immediately join the cluster in place of the missing broker with the same partitions and topics assigned to it



Cluster Membership

Whenever a broker is started, it registers with the ZooKeeper as shown in the below image:

```
[2017-11-10 03:04:44,816] INFO Accepted socket connection from /127.0.0.1:51906
(org.apache.zookeeper.server.NIOServerCnxnFactory)
[2017-11-10 03:04:44,817] INFO Client attempting to establish new session at /12
7.0.0.1:51906 (org.apache.zookeeper.server.ZooKeeperServer)
[2017-11-10 03:04:44,863] INFO Established session 0x15fa2a80e930006 with negoti
ated timeout 6000 for client /127.0.0.1:51906 (org.apache.zookeeper.server.ZooKe
eperServer)
```

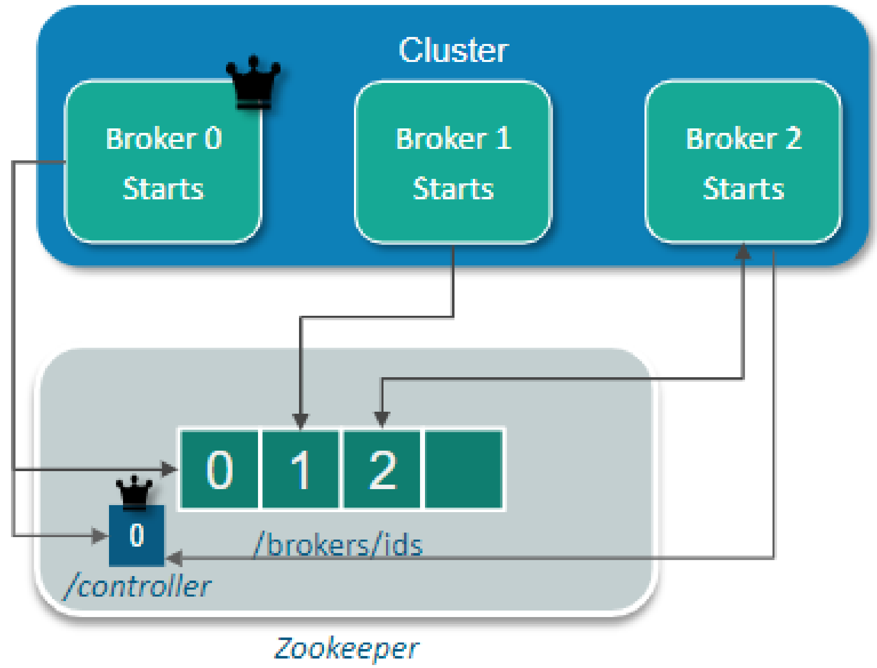
If we look at the /brokers/id in ZooKeeper, we will get the list of Brokers that are active:

```
[edureka@localhost kafka_2.12-0.11.0.0]$ ./bin/zookeeper-shell.sh localhost:218
 <<< "ls /brokers/ids"
Connecting to localhost:2181
Welcome to ZooKeeper!
JLine support is disabled

WATCHER::

WatchedEvent state:SyncConnected type:None path:null
[0, 1, 2]
```

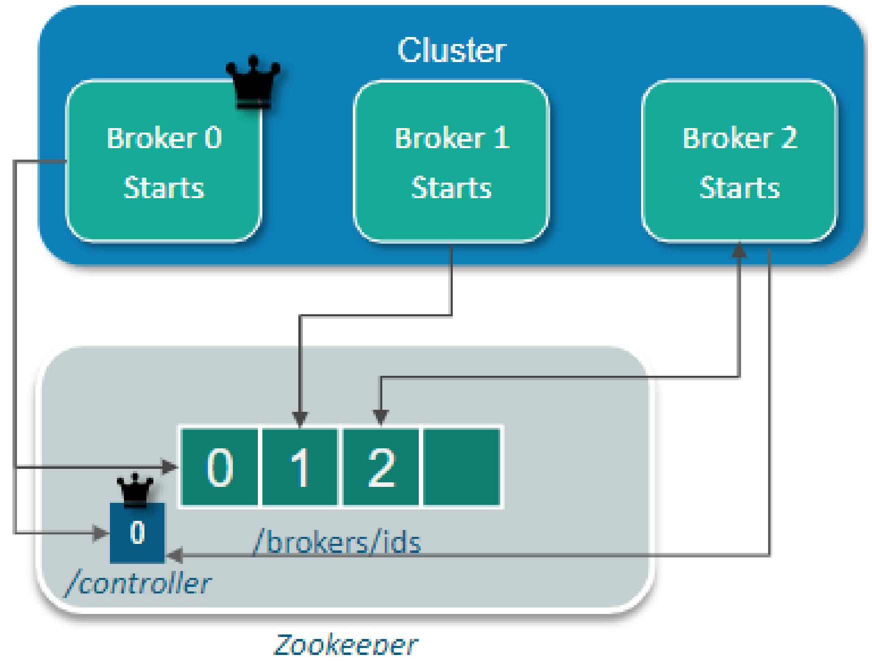
Let’s understand How Kafka Cluster Controller is elected

Controller

- The controller is one of the Kafka brokers which along with broker functionalities, is responsible for electing partition leaders
- The first broker that starts in the cluster becomes the controller by creating an ephemeral node in ZooKeeper called */controller*



Controller

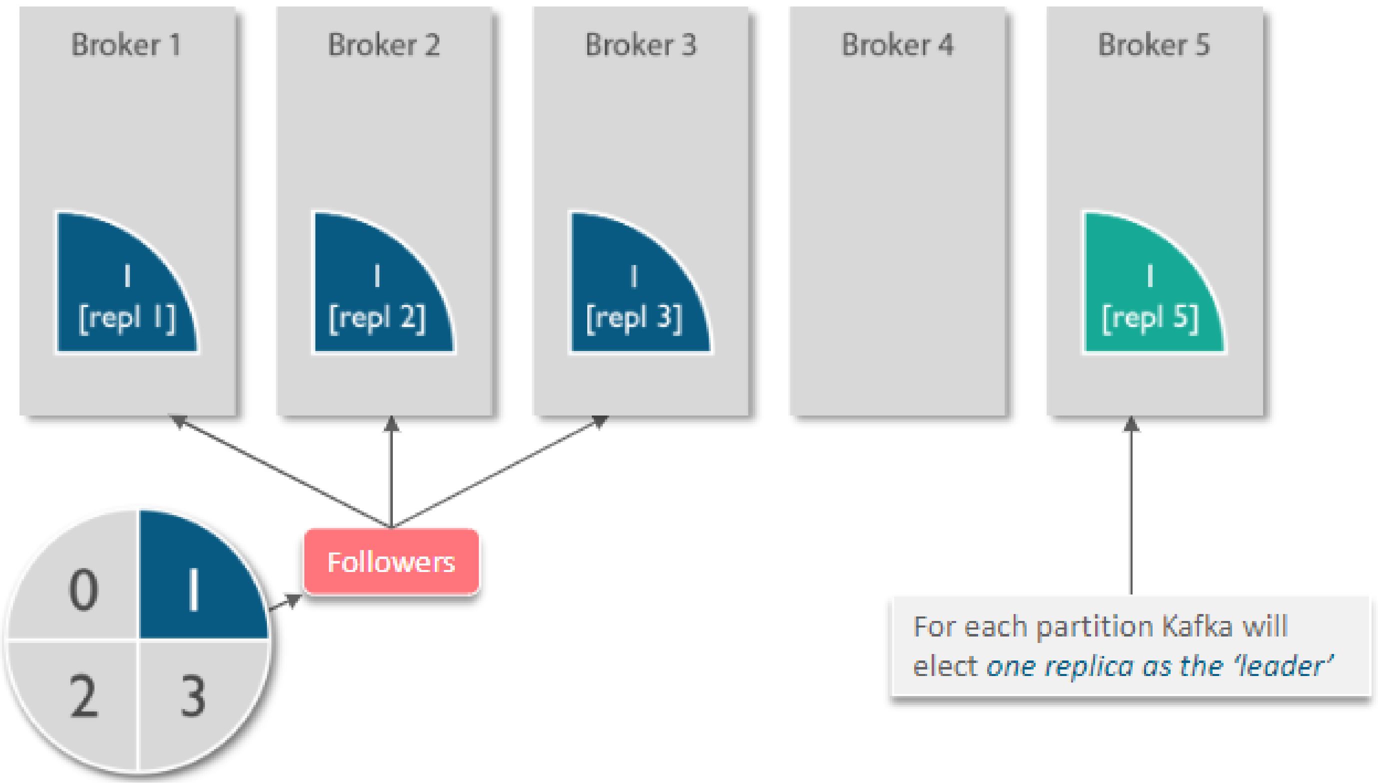


- When other brokers start, they also try to create this node, but receive a "node already exists" exception, which causes them to "realize" that the controller node already exists and that the cluster already has a controller
- The brokers create a *Zookeeper watch* on the controller node so they get notified of changes to this node

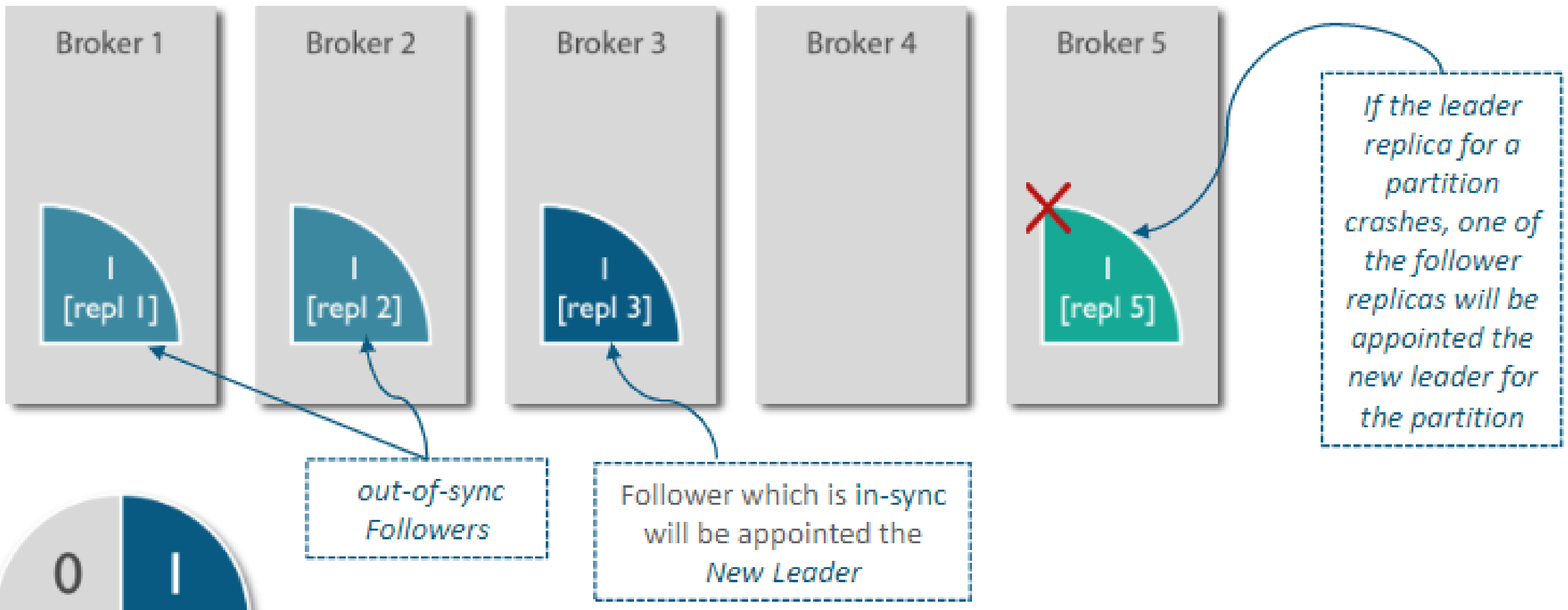
This way, we guarantee that the cluster will only have one controller at a time

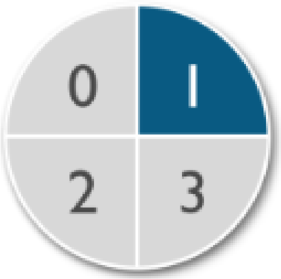
Let’s understand Kafka Replication Concepts

Replication



Replication

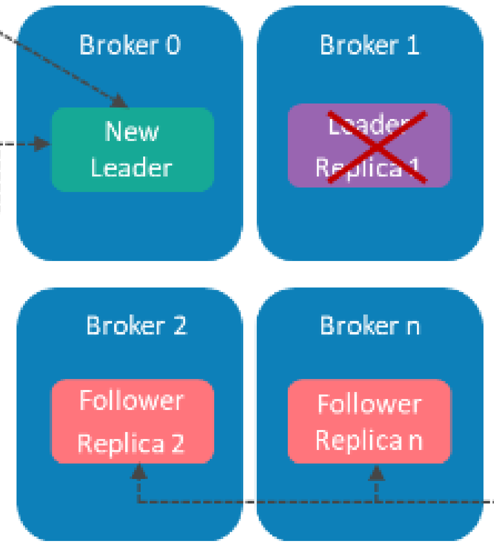




- Another task the leader is responsible for is knowing which of the follower replicas is up-to-date with the leader
- Followers attempt to stay up-to-date by replicating all the messages from the leader as the messages arrive, but they can fail to stay in-sync

Types of Replicas

- Each partition has a single replica designated as the *leader*
- All produce and consume requests go through the leader, in order to guarantee consistency



- All replicas for a partition that are not leaders are called *followers*
- Their only job is to replicate messages from the leader and stay *up-to-date* with recent messages

`replica.lag.time.max.ms` : The amount of time a follower can be inactive or behind before it is considered out of sync

Types of Replicas

This image shows partitions of a topic, leader of that partition, their replicas and in-sync replicas

```
[edureka@localhost kafka_2.12-0.11.0.0]$ bin/kafka-topics.sh --describe --zookeeper loc
alhost:2181 --topic edureka
Topic:edureka   PartitionCount:3        ReplicationFactor:3     Configs:
        Topic: edureka  Partition: 0    Leader: 0       Replicas: 0,1,2 Isr: 0,1,2
        Topic: edureka  Partition: 1    Leader: 1       Replicas: 1,2,0 Isr: 1,2,0
        Topic: edureka  Partition: 2    Leader: 2       Replicas: 2,0,1 Isr: 2,0,1
```

When Broker 1 gets down, the replica that is in-sync with it becomes out-of-sync

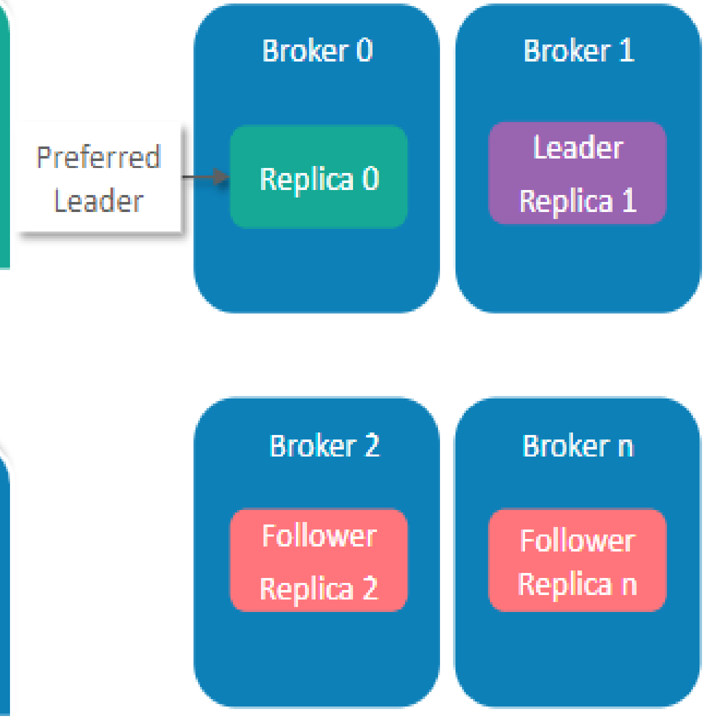
```
[edureka@localhost kafka_2.12-0.11.0.0]$ bin/kafka-topics.sh --describe --zookeeper loc
alhost:2181 --topic edureka
Topic:edureka   PartitionCount:3        ReplicationFactor:3     Configs:
        Topic: edureka  Partition: 0    Leader: 0       Replicas: 0,1,2 Isr: 0,2
        Topic: edureka  Partition: 1    Leader: 2       Replicas: 1,2,0 Isr: 2,0
        Topic: edureka  Partition: 2    Leader: 2       Replicas: 2,0,1 Isr: 2,0
```

Only in-sync replicas are eligible to be elected as partition leaders in case the existing leader fails

Preferred Leader

1. Each partition has a ***preferred leader***— The replica that was the leader when the topic was originally created

2. It is preferred because when partitions are first created, the leaders are balanced between brokers



3. As a result, we expect that when the preferred leader is indeed the leader for all partitions in the cluster, load will be evenly balanced between brokers

4. `auto.leader.rebalance.enable=true` : In case of failure of leader if the preferred leader replica is in-sync then it triggers leader election to make the preferred leader the current leader

Let’s see how Requests work in Kafka

Requests - Format

Request Header

- Request Type
- Correlation ID
- Request Version
- Client ID

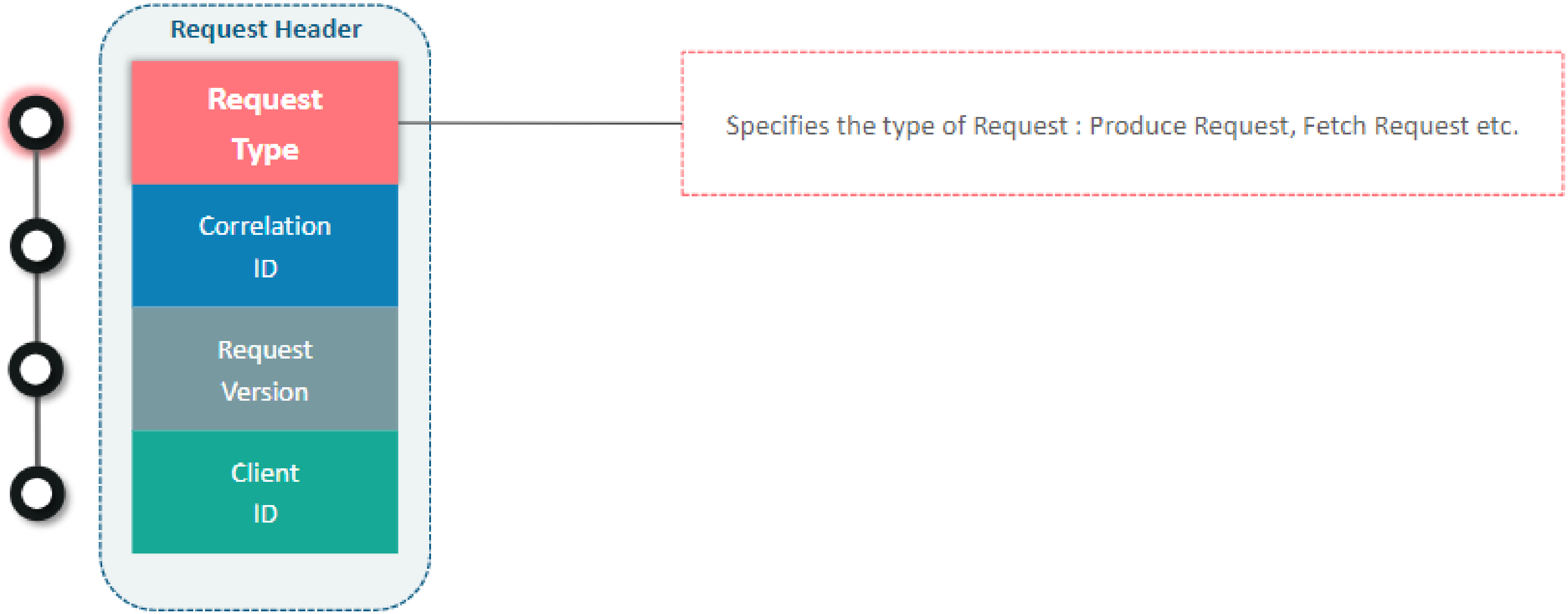
- Kafka has a *binary request format*
- It is applicable in both cases, when the request is processed successfully or when the broker encounters errors while processing the request
- Clients always initiate connections and send requests, and the broker processes the requests and responds to them

Request

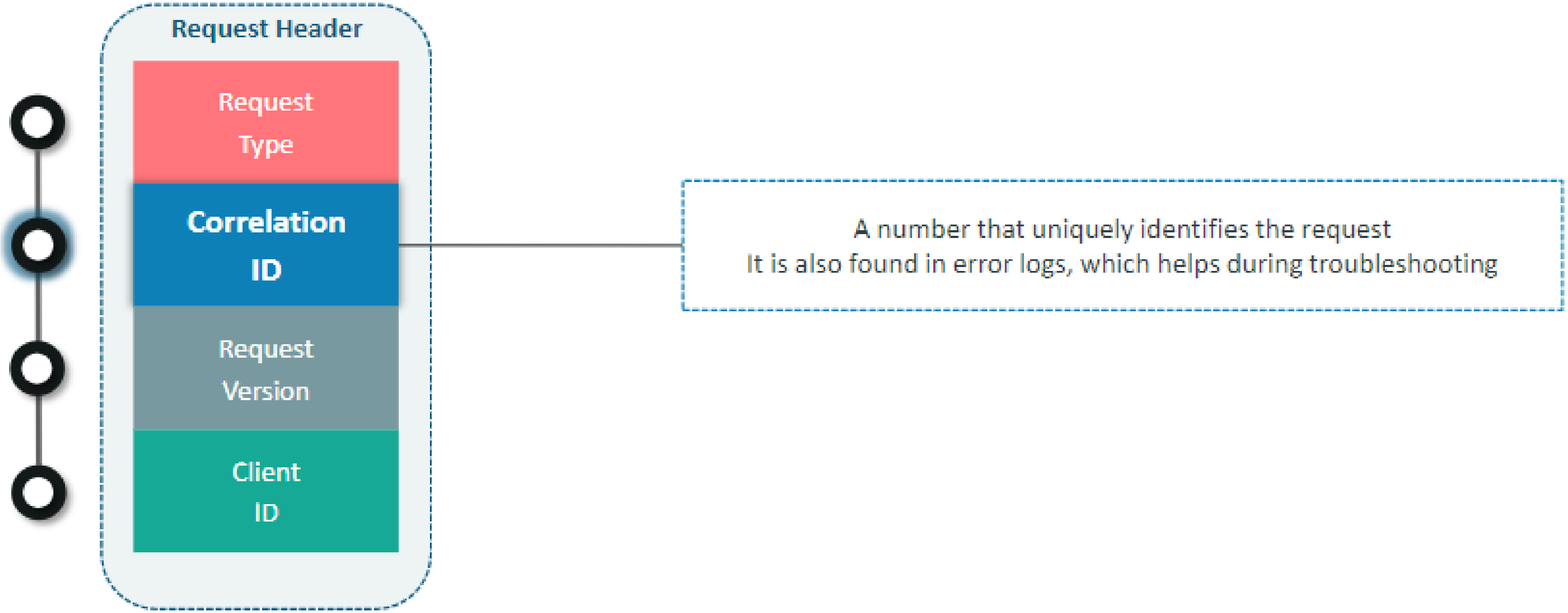
Header	Data

Kafka Header Comprises of different Parameters

Request - Header - Request Type

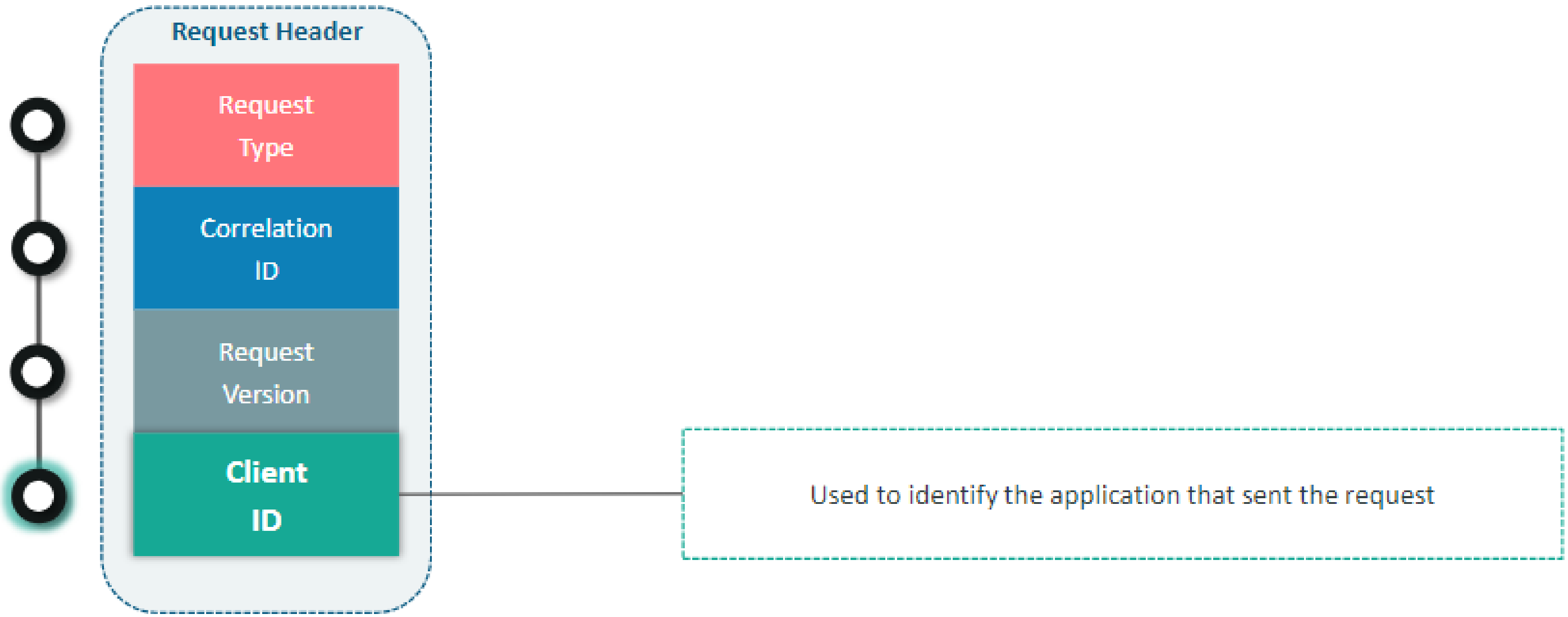


Request - Header - Correlation ID



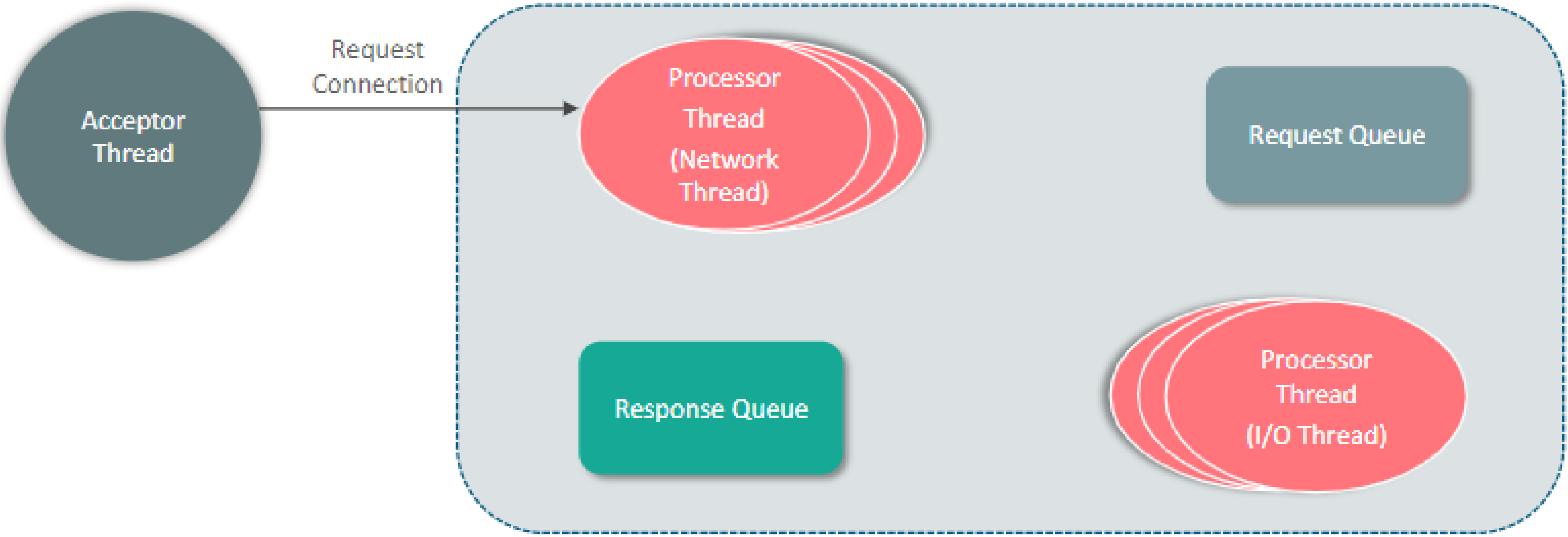
Request - Header - Request Version

Request - Header - Client ID



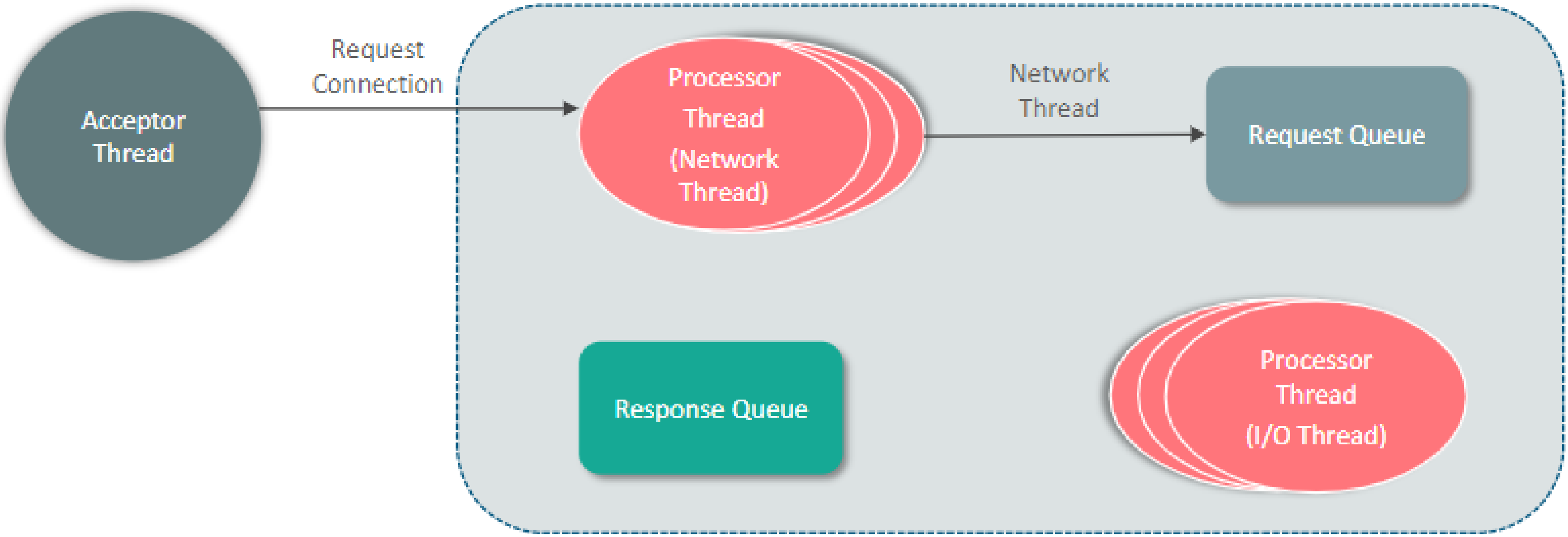
Request Processing

The broker *acceptor thread* creates a connection and hands it over to a processor thread



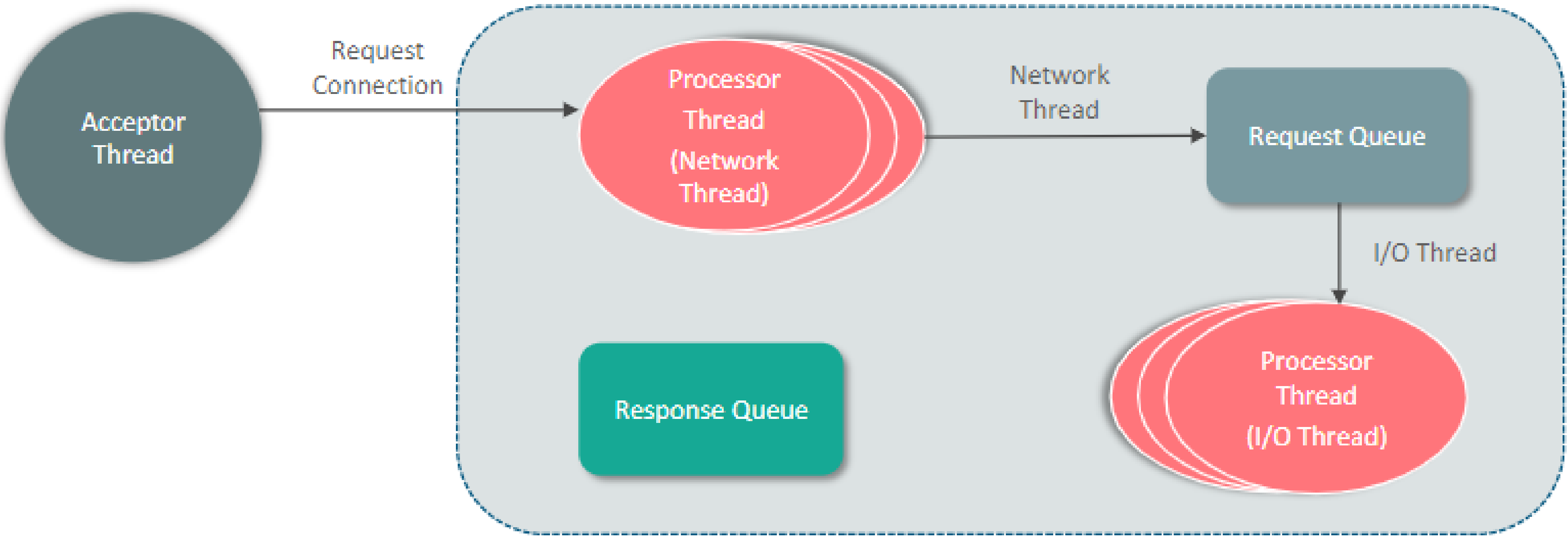
Request Processing

The *network threads* are responsible for taking requests from client connections, placing them in a request queue



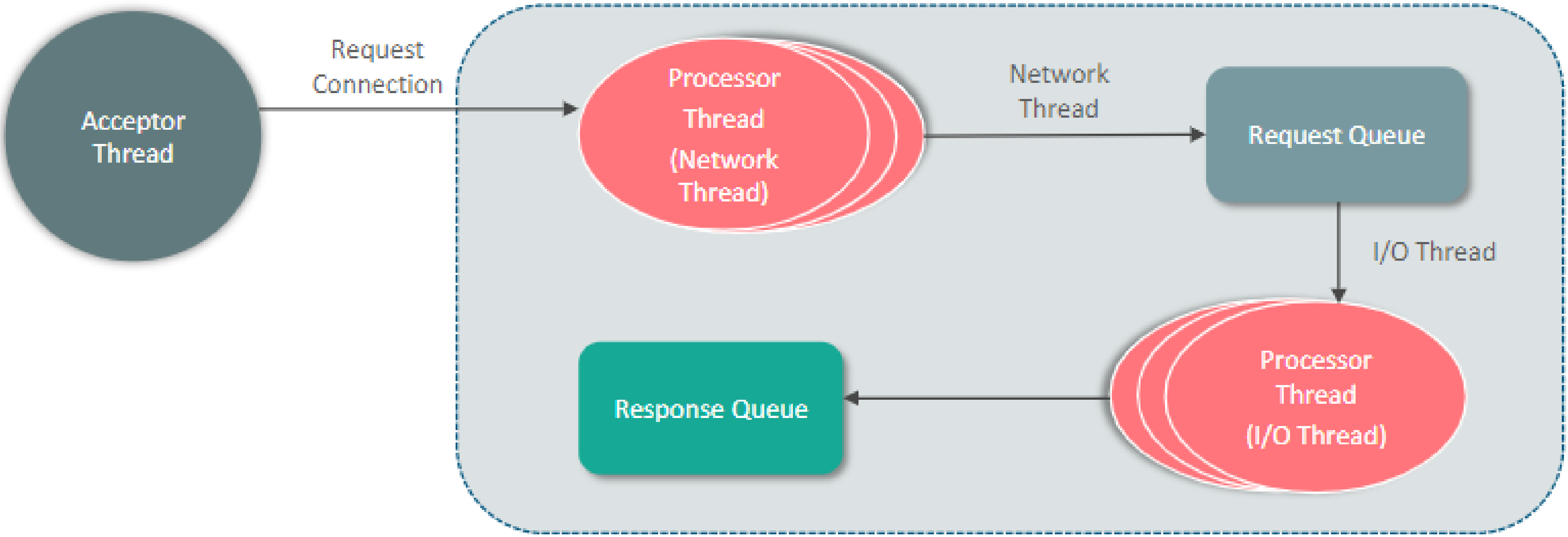
Request Processing

Once requests are placed on the request queue, *I/O threads* are responsible for picking them up and processing them



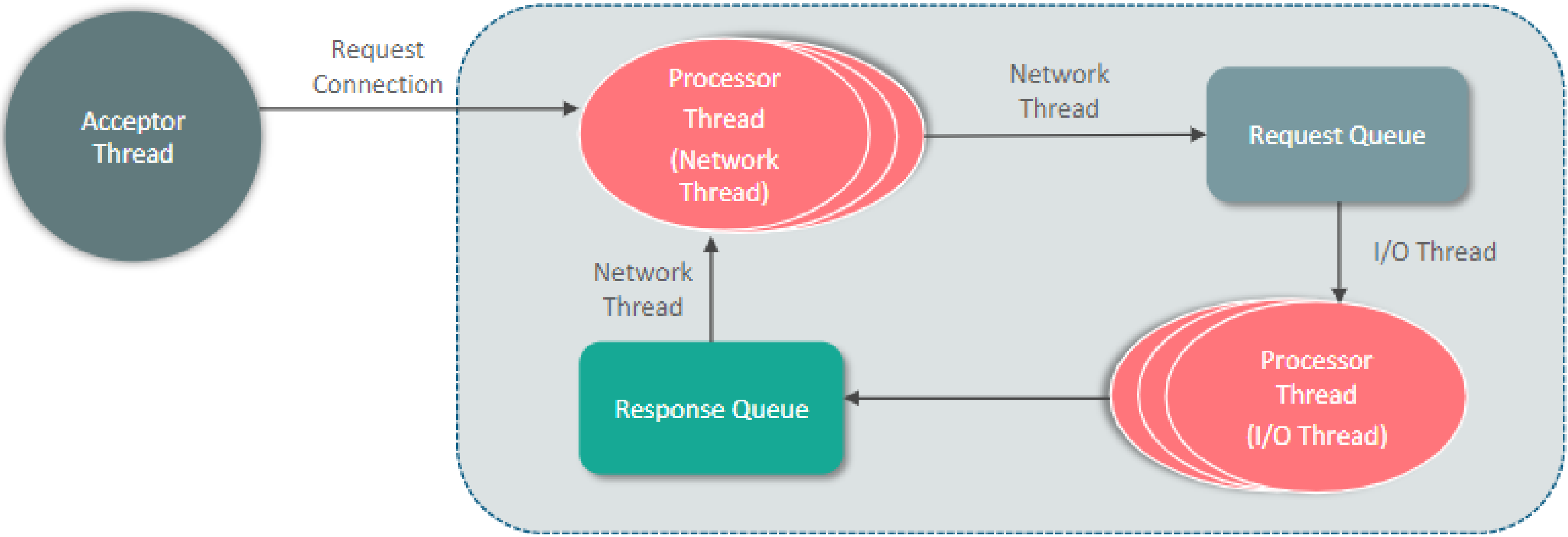
Request Processing

The Processed messages are sent to the Response Queue



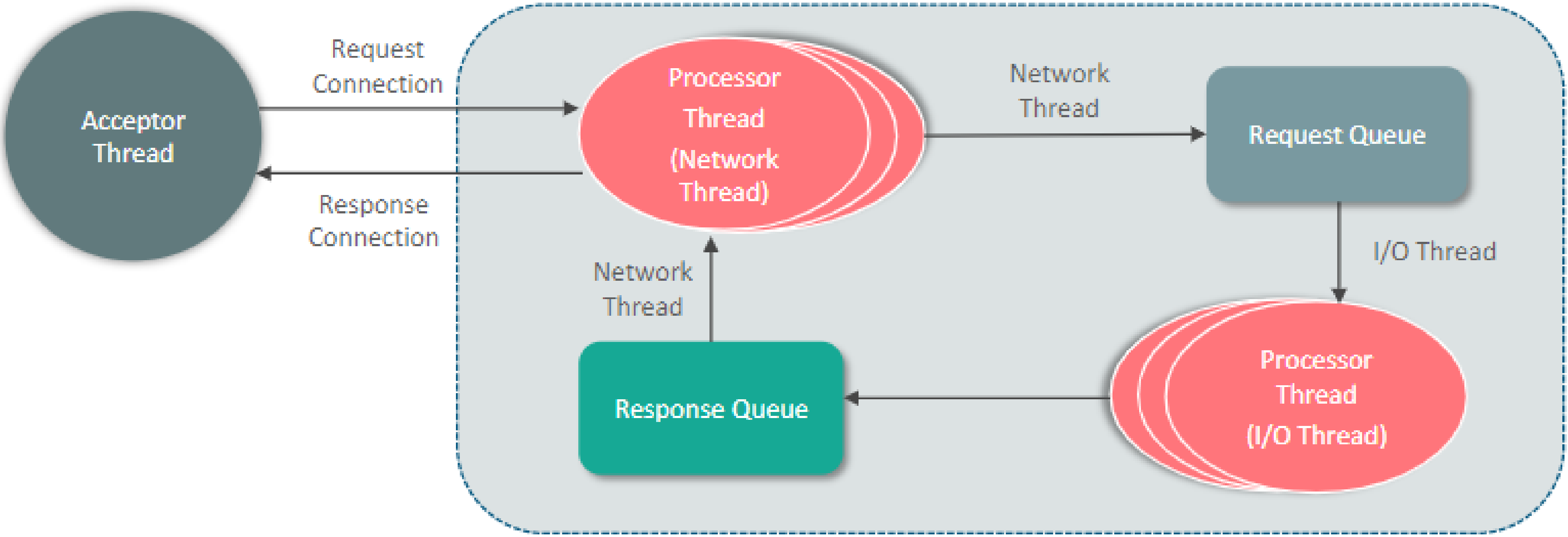
Request Processing

The *network threads* are also responsible for picking up responses from a response queue and sending them back to clients



Request Processing

This response is then sent to the Client



Types of Requests

Produce Request

Fetch Request

Let's see the validations required for Produce Requests

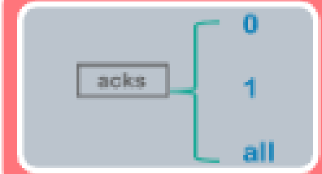
Produce Request

Fetch Request

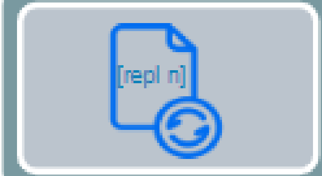
Validations required for Produce Request

When the broker that contains the lead replica for a partition receives a produce request for this partition, it will start by running a few validations:

Does the user who is sending the data have **write** privileges on the topic?



Is the number of "acks" specified in the request valid? (only 0, 1, and "all" are allowed)



If "acks" is set to **all**, are there enough in-sync replicas for safely writing the message?

Produce Requests

Once the validations are performed, Produce Request will write the new messages to local disk

Kafka does not wait for the data to get persisted to disk—it relies on replication for message durability

Once the message is written to the leader of the partition, the broker examines the "acks" configuration—if "acks" is set to 0 or 1, the broker will respond immediately

If "acks" is set to all, the request will be stored in a buffer called ***purgatory*** until the leader observes that the follower replicas replicated the message, at which point a response is sent to the client

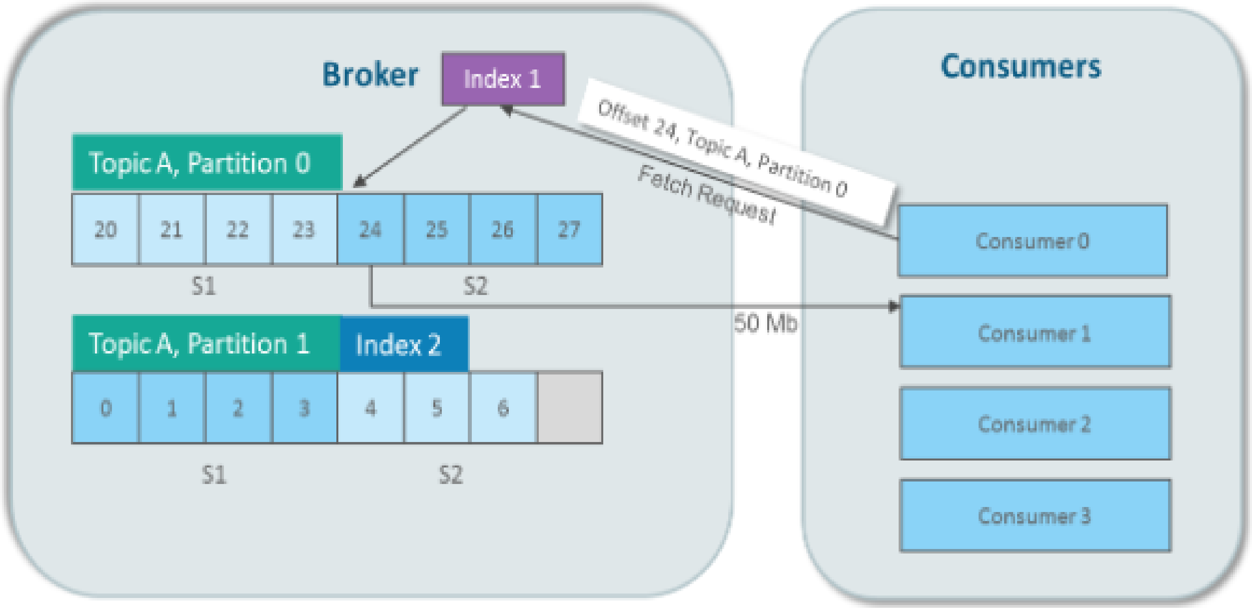
Let’s know about Fetch Requests

Produce Request

Fetch Request

Fetch Requests

Brokers process fetch requests in a way that is very similar to the way produce requests are handled



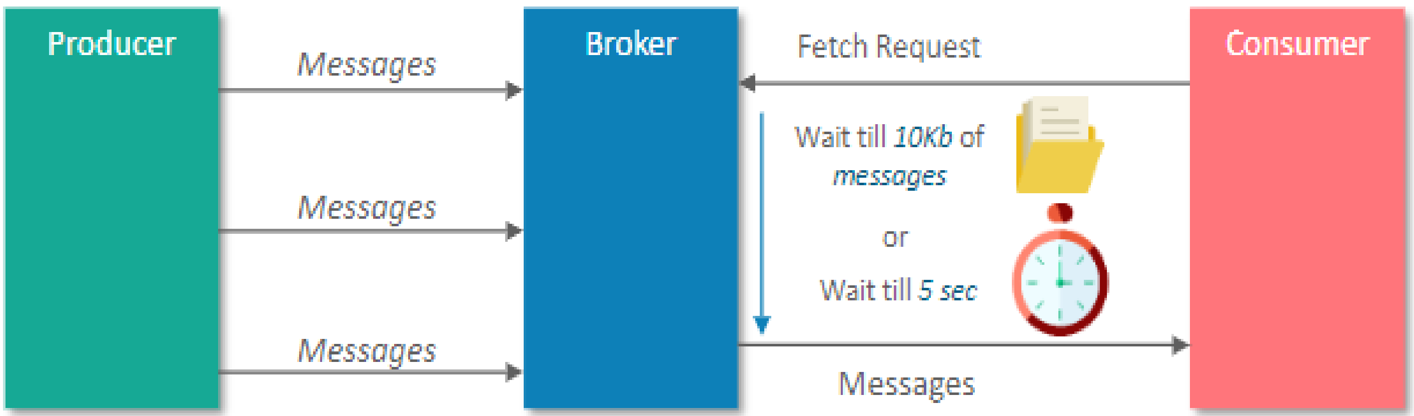
The client sends a request, asking the broker to send messages from a list of topics, partitions, and offsets

- Clients also specify a limit to how much data the broker can return for each partition
- This limit is mandatory, if not, the broker could send huge amount of data, which causes the consumer to go out of memory

Fetch Requests

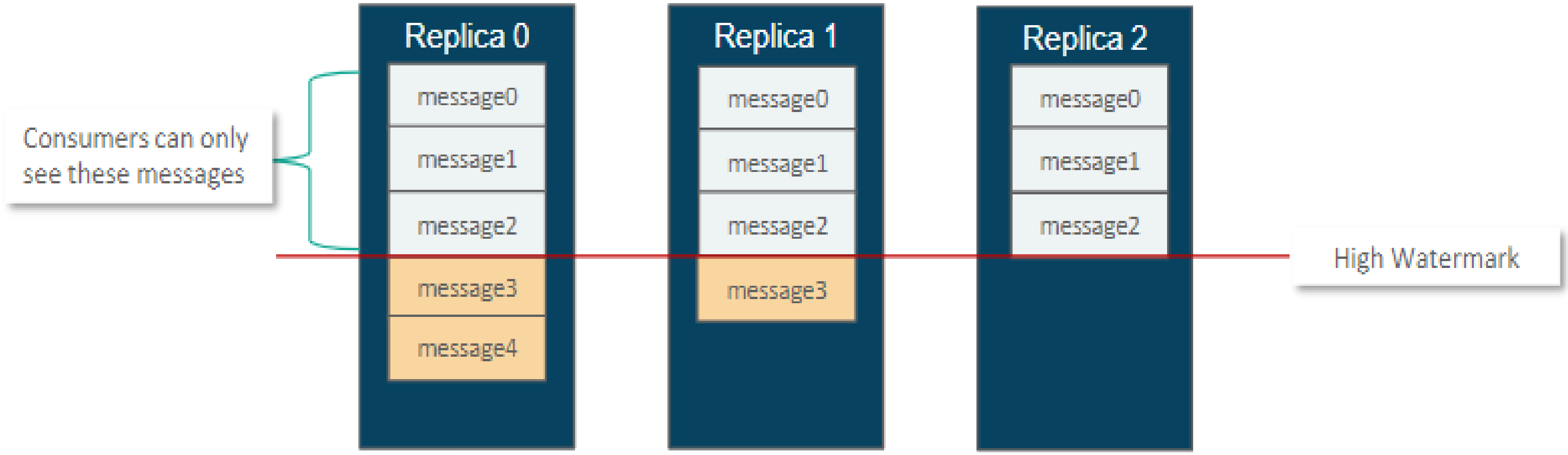
Consumer *configures* a *Lower Boundary*, which *specifies* the *minimum amount* of *data* should be *present before sending*, thus helps network utilization

Clients can also define a *timeout* to *tell* the *broker* "*If* you *didn't receive* the *minimum amount* of *data* to *send within x milliseconds*, just *send what you have*"

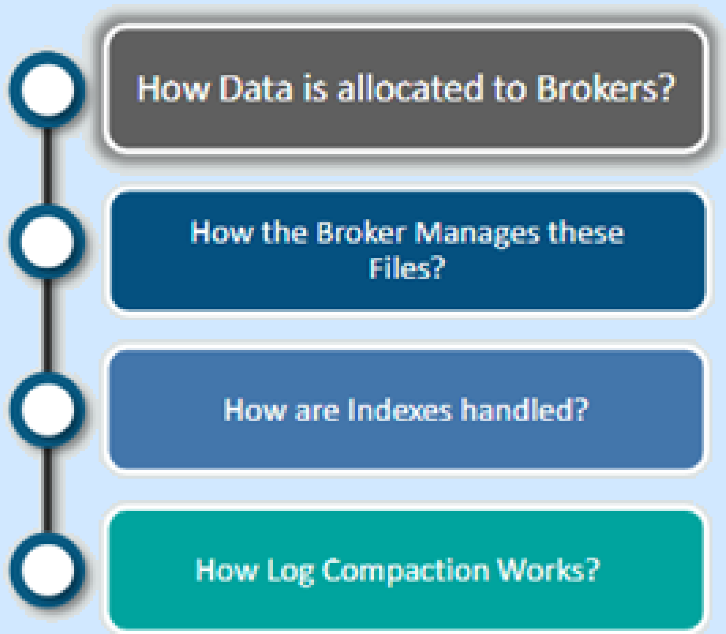


Fetch Requests

- Consumers can only read messages that were written to all in-sync replicas
- If your in-sync replicas have not received the latest messages they will get an empty response
- *replica.lag.time.max.ms*—The maximum time given to a replica for replicating new messages while still being considered in-sync



Physical Storage



Let’s see how data is stored in Kafka

Partition Allocation

0 1 2 3 4 5 6 7 8 9

Topic

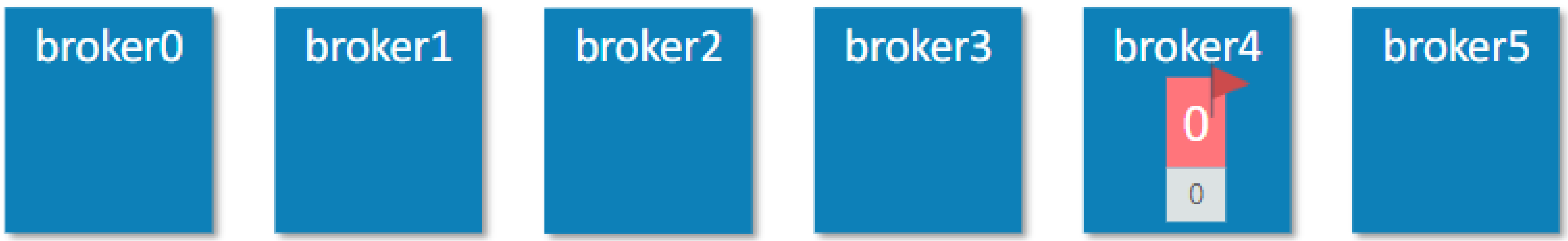
Consider a topic with 10 partitions, Replication factor for each partition is 3, thus we have to allocate 3X10=30 replicas

broker0 broker1 broker2 broker3 broker4 broker5

We have 30 replicas which are to be allocated on 6 Brokers evenly
Thus, 30/6 = 5 partitions/broker will be allocated

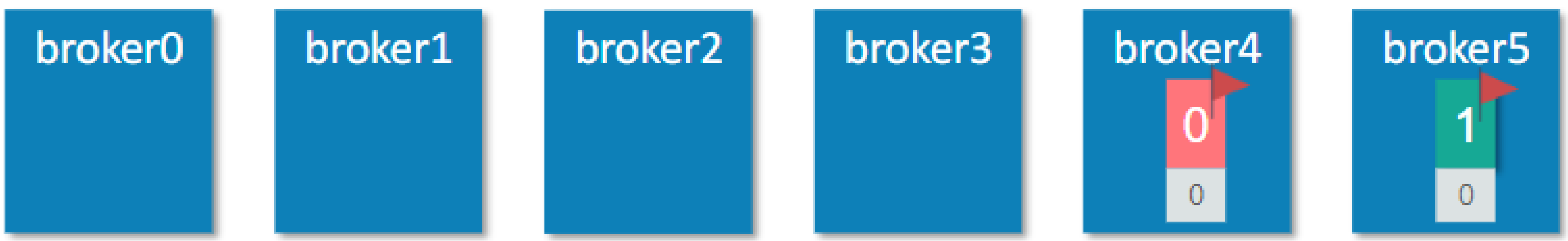
Partition Allocation - Leader

We start by allocating the leader of partitions
Let's consider the leader for *partition 0* will be allocated on *broker4*

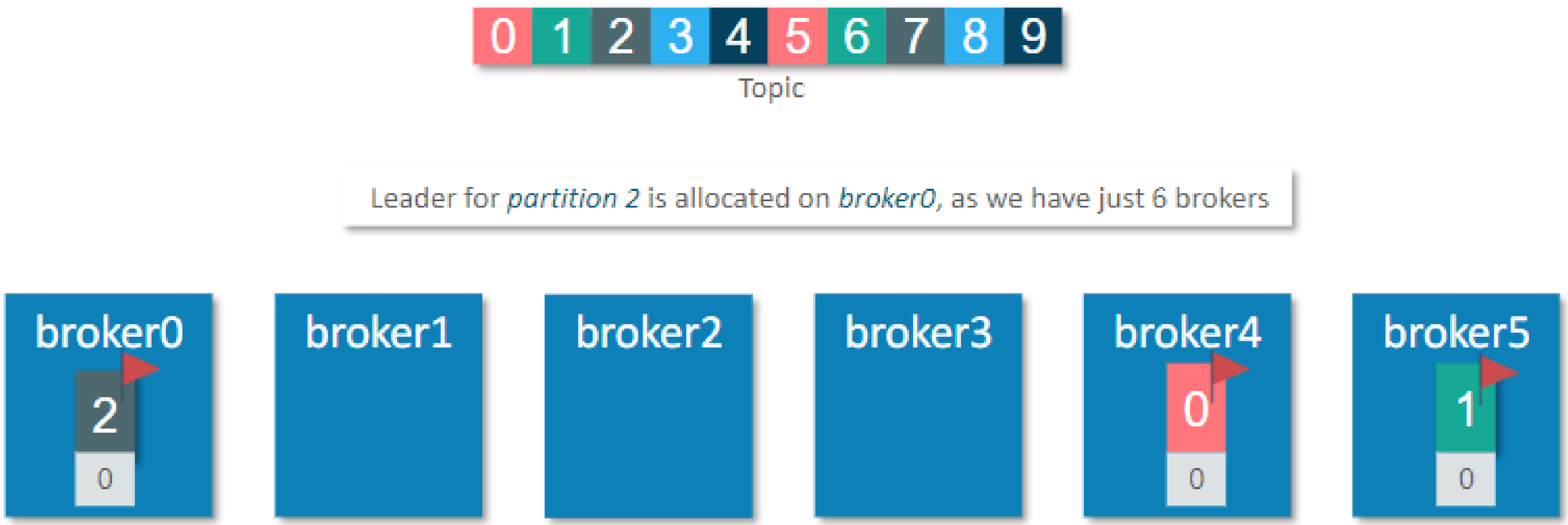


Partition Allocation - Leader

Leaders will be allocated according to the *Round-Robin algorithm*
Thus, Leader for partition 1 will be allocated on *broker5*

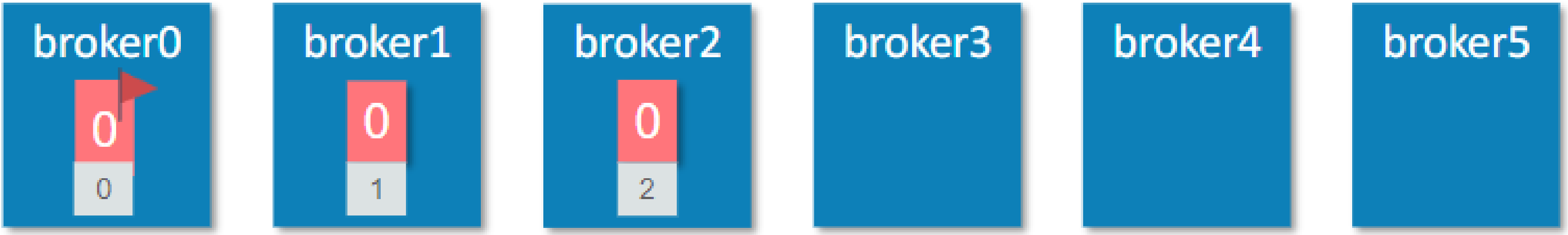


Partition Allocation - Leader



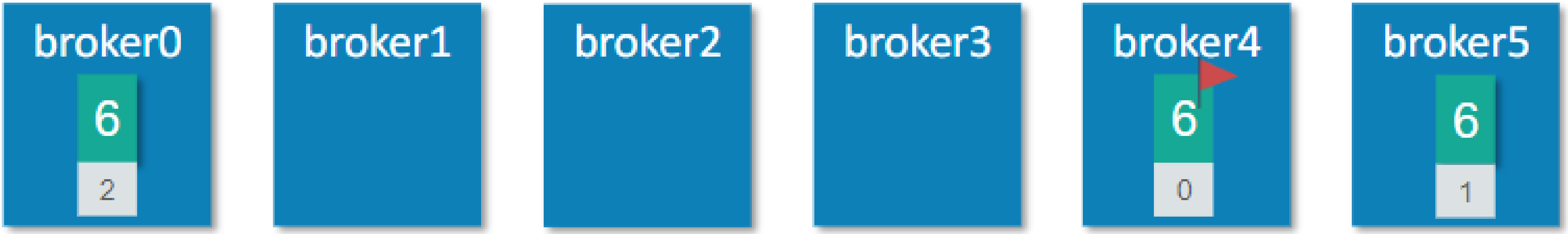
Partition Allocation - Follower

If the leader for *partition 0* is placed on *broker0*, we can allocate the followers in increasing offset on *broker1* and *broker2*, but not on *broker 0* or both the followers on *broker 1*



Partition Allocation - Follower

Similarly, if the leader for *partition 6* is on the *broker 4*,then its follower replicas must be placed at increasing offsets from the leader on *broker 5* and *broker 0*



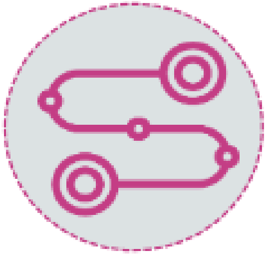
File Management

Retention

- Kafka does not keep data forever, nor does it wait for all consumers to read a message before deleting it
- Kafka administrator configures a *retention period* for each topic—
  1. *Amount of time* to store messages before deleting them
  2. *Amount of data* to store before older messages are purged

Segments

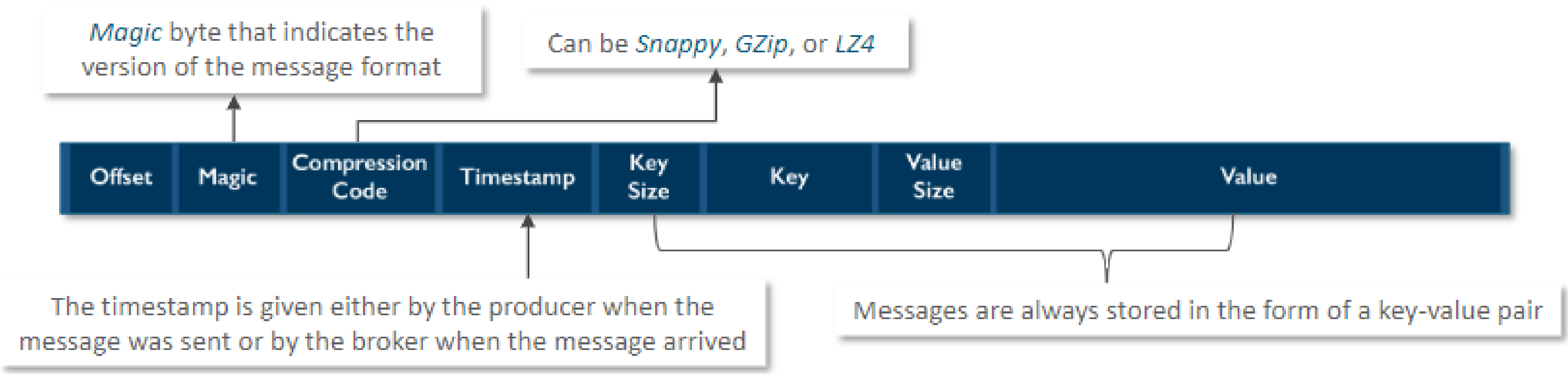
- Finding the messages that needs to be removed in a large file and then deleting a portion of the file is both time-consuming and error-prone. Thus, each partition splits into *segments*
- By default, each segment contains either 1 GB of data or a week of data, whichever is smaller as a Kafka broker is writing to a partition
- If the segment limit is reached, we close the file and start a new one



- How Data is allocated to Brokers?
- How the Broker Manages these Files?
- How are Indexes handled?
- How Log Compaction Works?

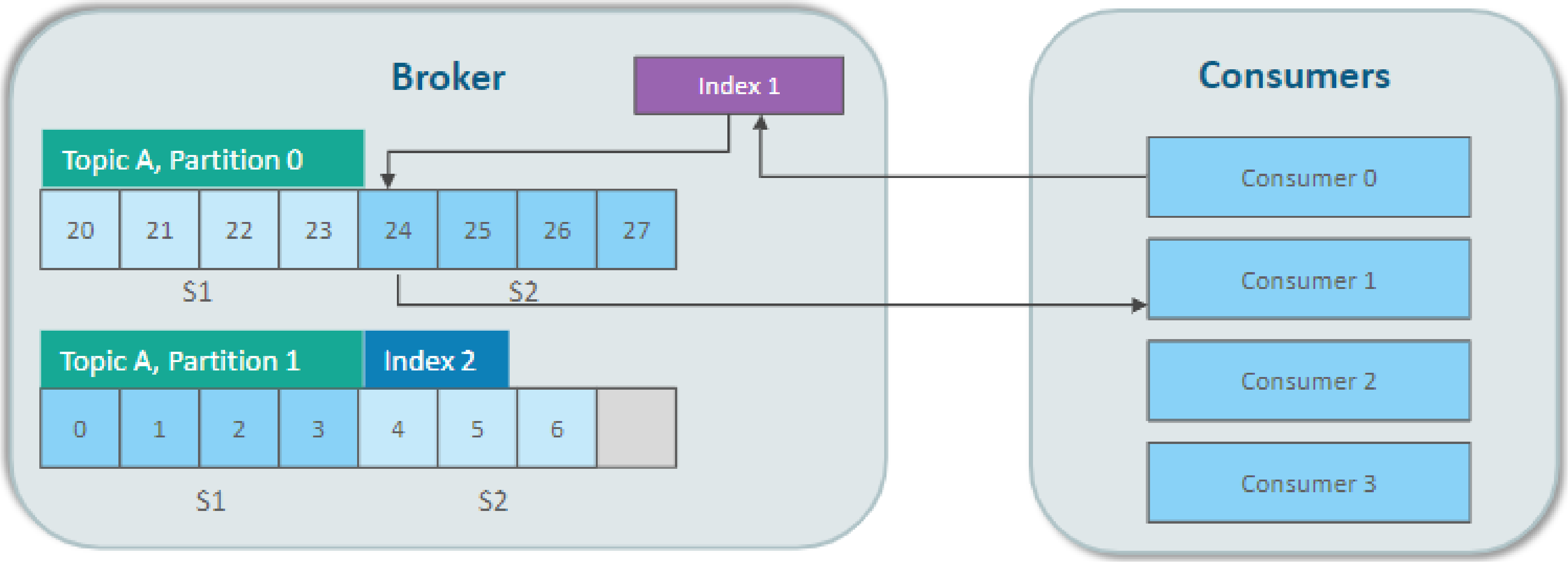
File Format

- Each segment is stored in a single data file
- The format of the data on the disk is identical to the format of the messages that we send
- Using the same message format on disk is what allows Kafka to use zero-copy optimization when sending messages to consumers



- How Data is allocated to Brokers?
- How the Broker Manages these Files?
- How are Indexes handled?
- How Log Compaction Works?

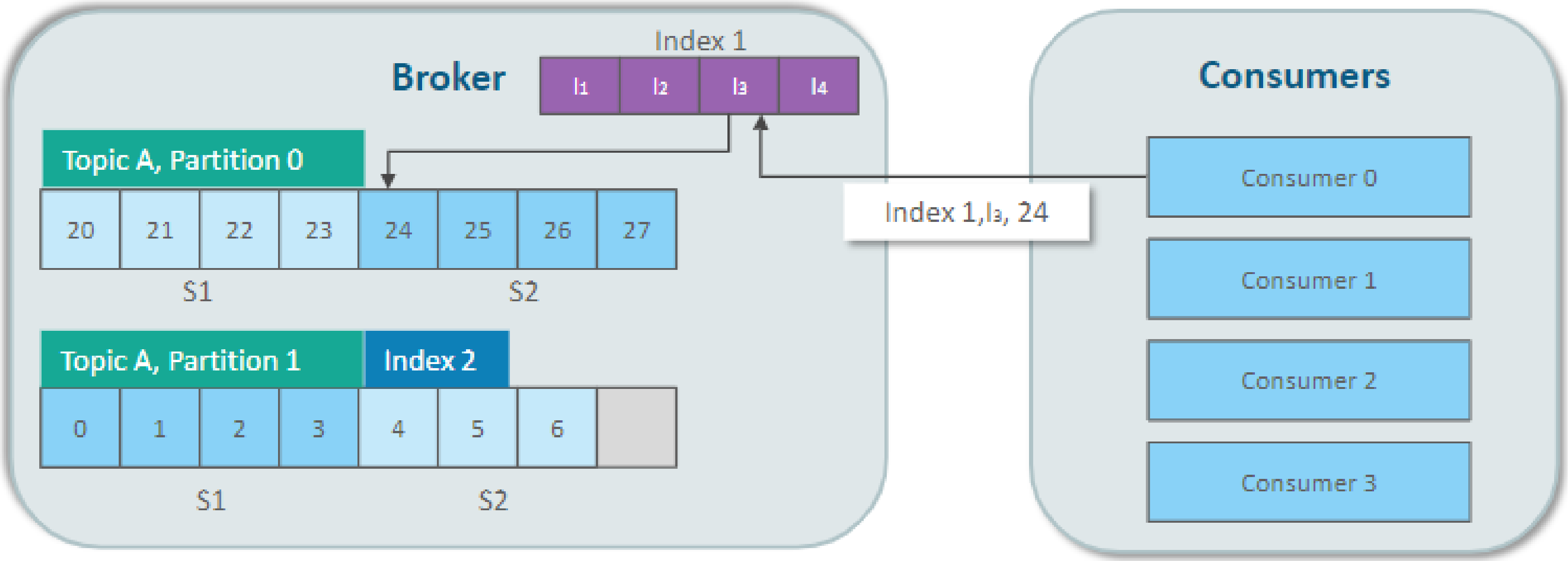
Indexes



Index : Kafka maintains an index for each partition, in order to help brokers quickly locate the message for a given offset

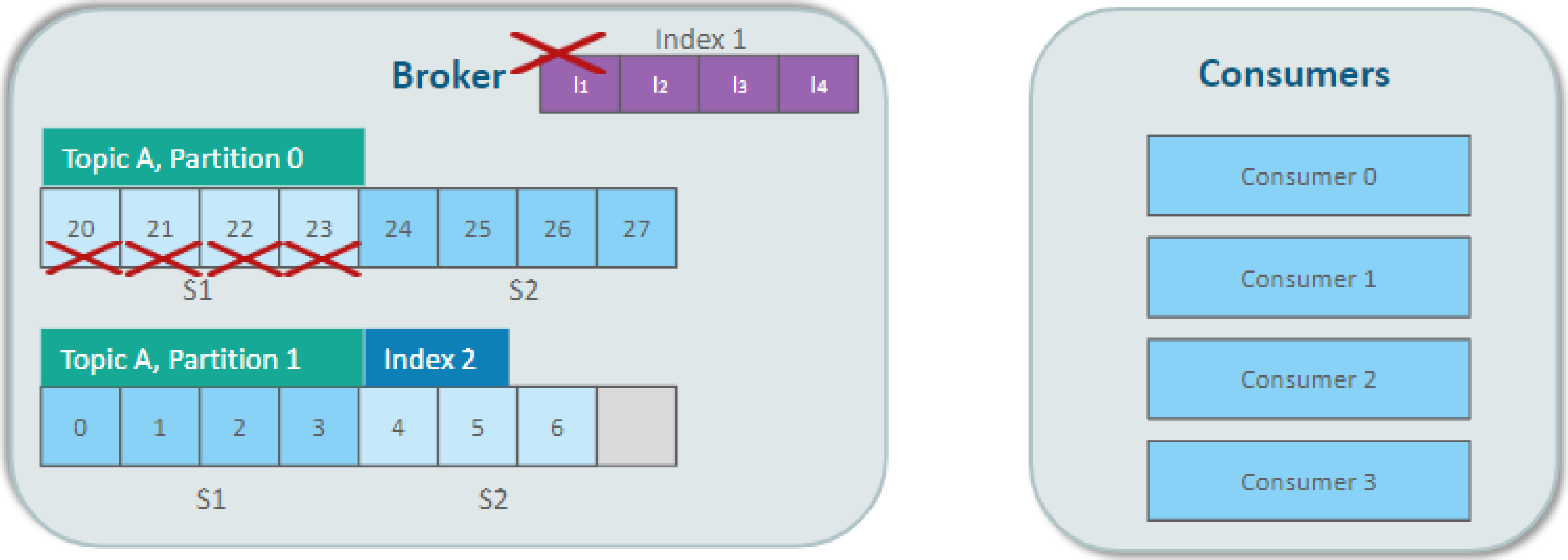
The index maps offsets to segment files and positions within the file, For example – Consumer 0 is fetching messages from offset 24,using Index 1

Indexes



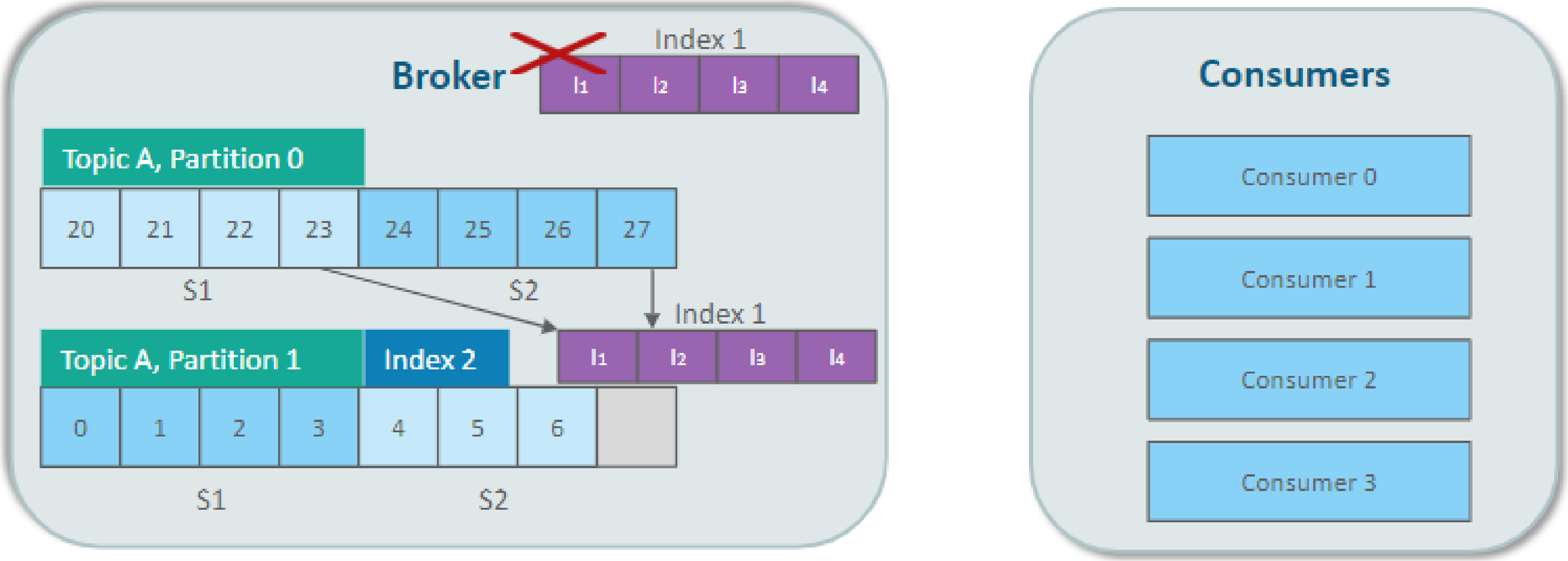
Indexes are also broken into segments(I1,I2,I3,I4)

Indexes

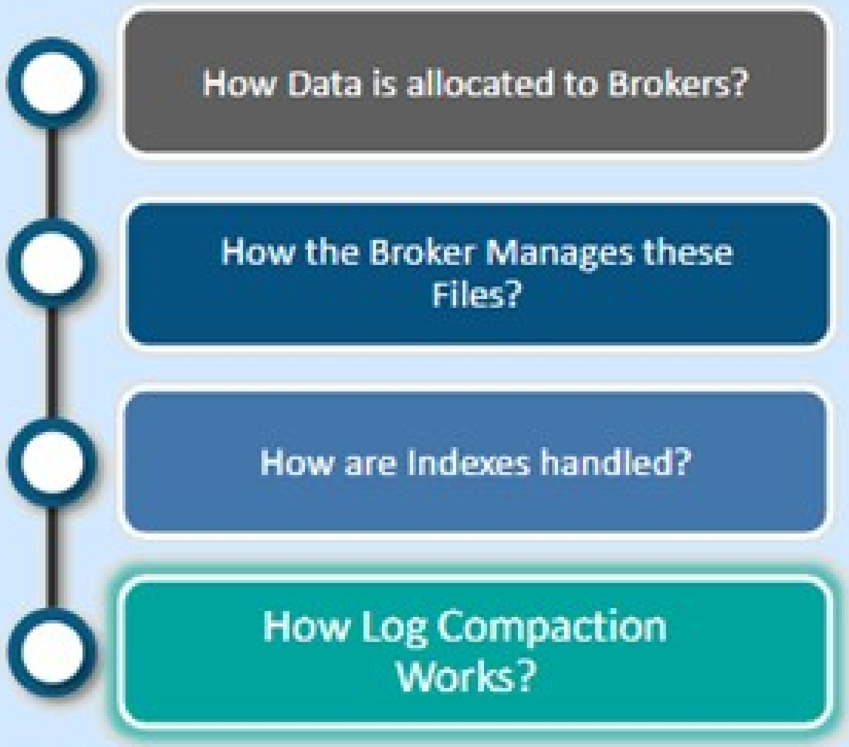


The system deletes index segment files along with in the log segments files

Indexes



If the index becomes corrupted, it will get regenerated from the matching log segment simply by re-reading the messages and recording the offsets and locations


How Data is allocated to Brokers?
How the Broker Manages these Files?
How are Indexes handled?
How Log Compaction Works?

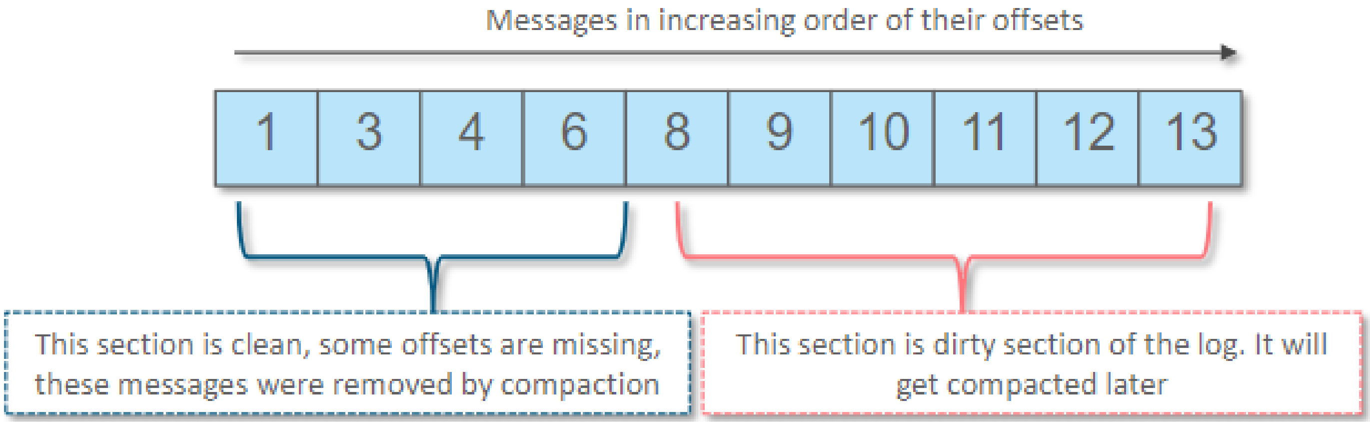
Clean and Dirty Messages

Clean

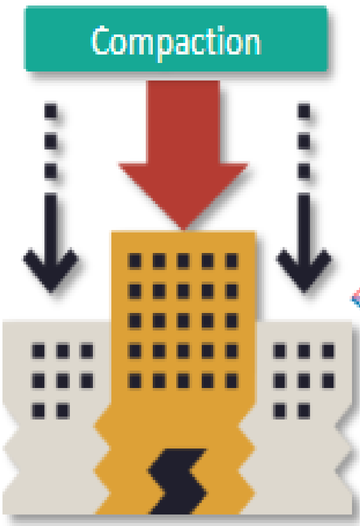
Messages that have been compacted before i.e. it contains only one value for each key, which is the latest value at the time of the pervious compaction

Dirty

Messages that were not compacted written after the last compaction



Compaction



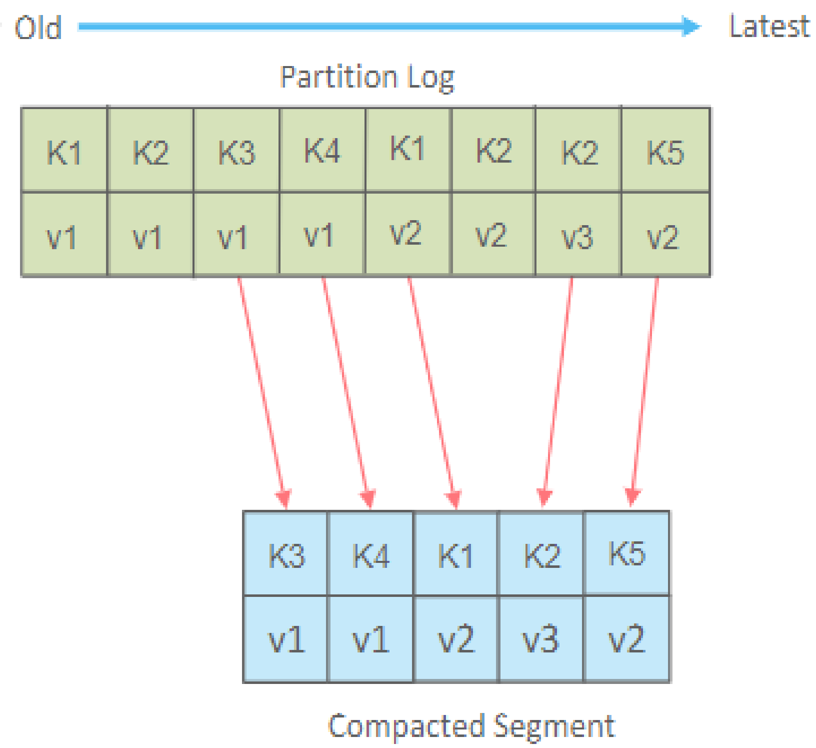
Setting the policy to compact only makes sense on topics for which applications produce events that contain both a key and a value

If retention is set to *delete*, it deletes events older than retention time, to compact, and only stores the most recent value for each key in the topic

If the topic contains *null keys*, compaction will fail

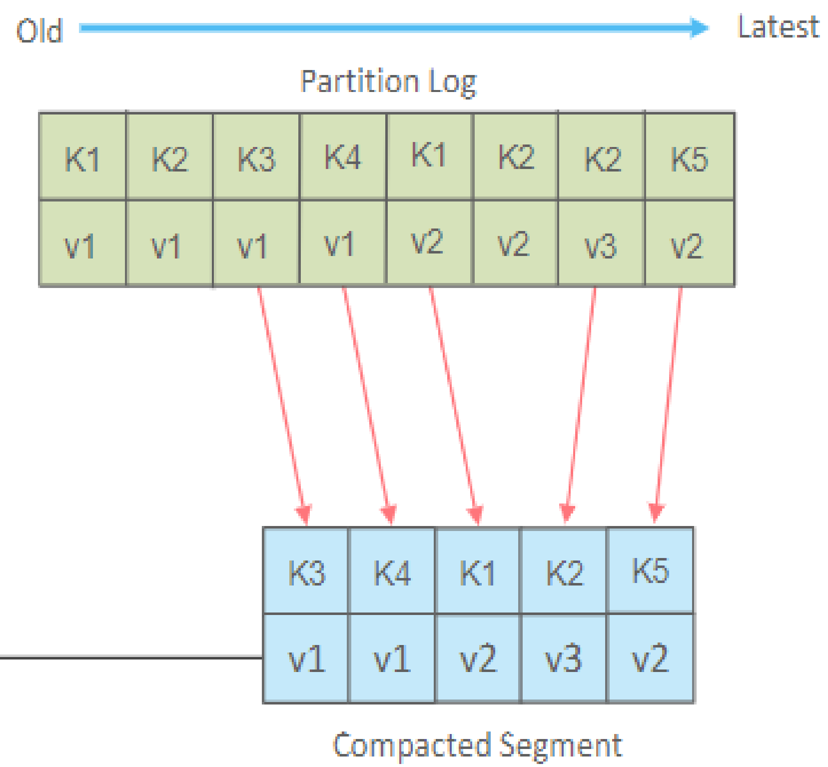
Compaction - Working

Once the cleaner thread builds the offset map, it will start reading off the clean segments, starting with the oldest, and check their contents against the offset map

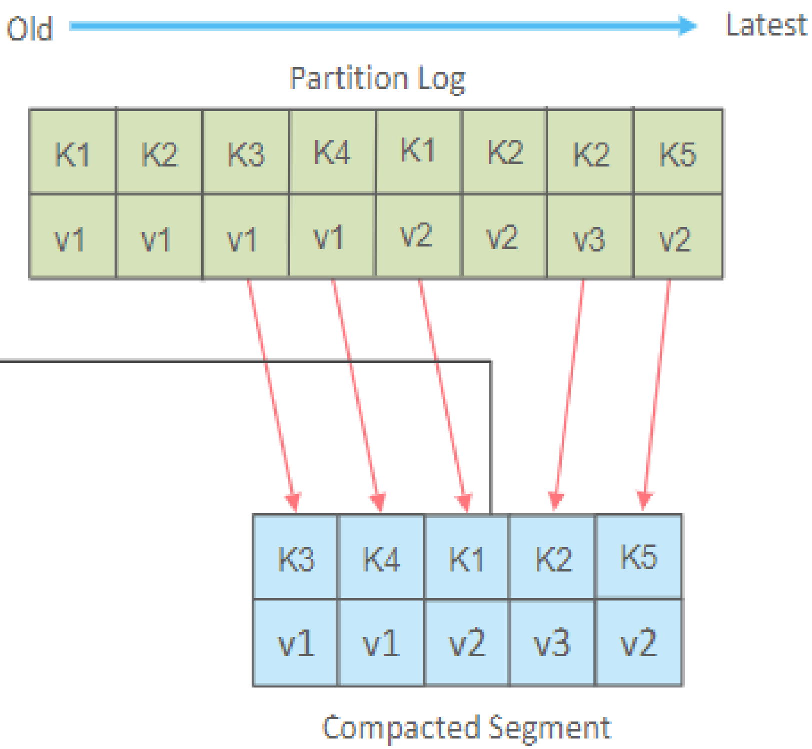


Compaction - Working

- It checks if the key of the message is present in the offset map
- If the key does not exist in the map, the value of the message we've just read is still the latest and we copy over the message to a replacement segment



Compaction - Working

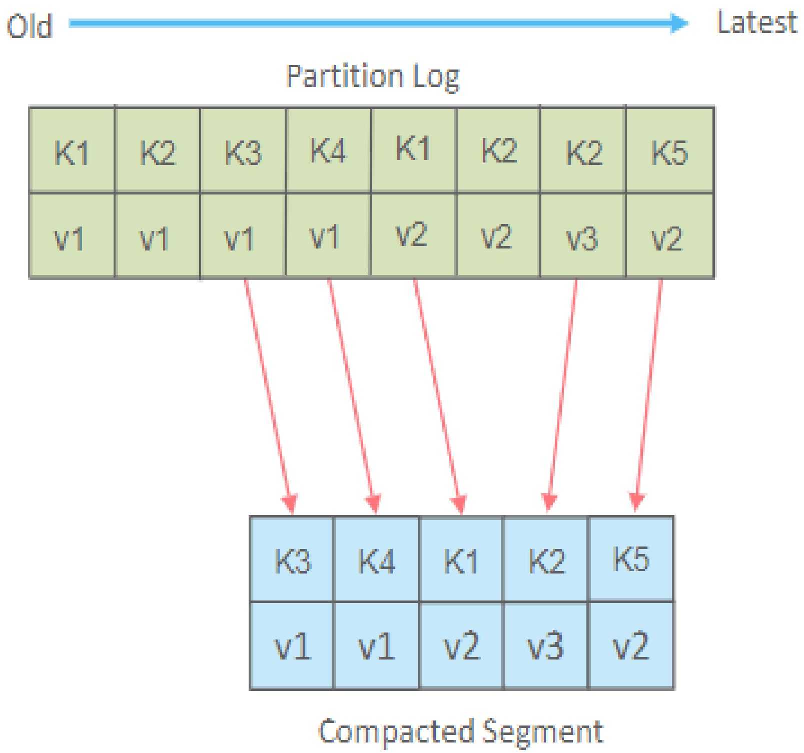


- If the key exists in the map, we omit the message because there is a message with an identical key but newer value later in the partition

- After copying all the messages that still contain the latest value for their key, we swap the replacement segment with the original and move on to the next segment

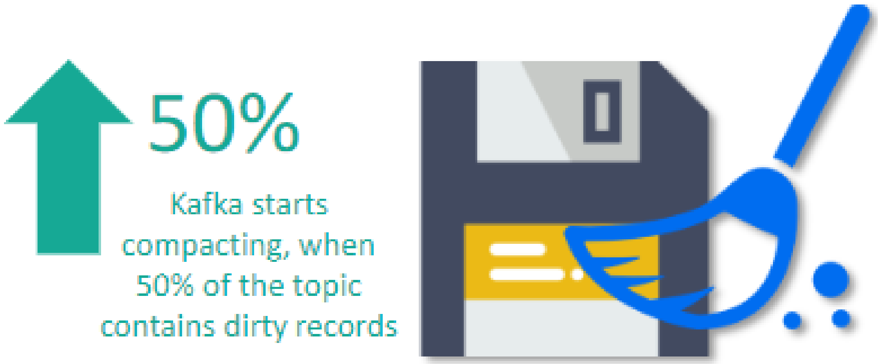
Compaction - Working

At the end of the process, we are left with one message per key—the one with the latest value



When are topics compacted ?

Only when segments become inactive, messages are eligible for compaction
Kafka starts compacting, when 50% of the topic contains dirty records



- Performing Compaction increases the number of reads and writes which has an impact on the performance
- We can not leave too many dirty data records as it consumes too much of disk-space
- Wasting 50% of disk-space used by a topic on dirty records and then compacting them in one go isa beneficial trade-off

Compacting topics

Property to compact topic

```
[edureka@localhost kafka_2.12-0.11.0.0]$ bin/kafka-topics.sh --zookeeper localhost:2181
 --alter --topic my-topic --config cleanup.policy=compact
WARNING: Altering topic configuration from this script has been deprecated and may be r
emoved in future releases.
         Going forward, please use kafka-configs.sh for this functionality
Updated config for topic "my-topic".
[edureka@localhost kafka_2.12-0.11.0.0]$
```

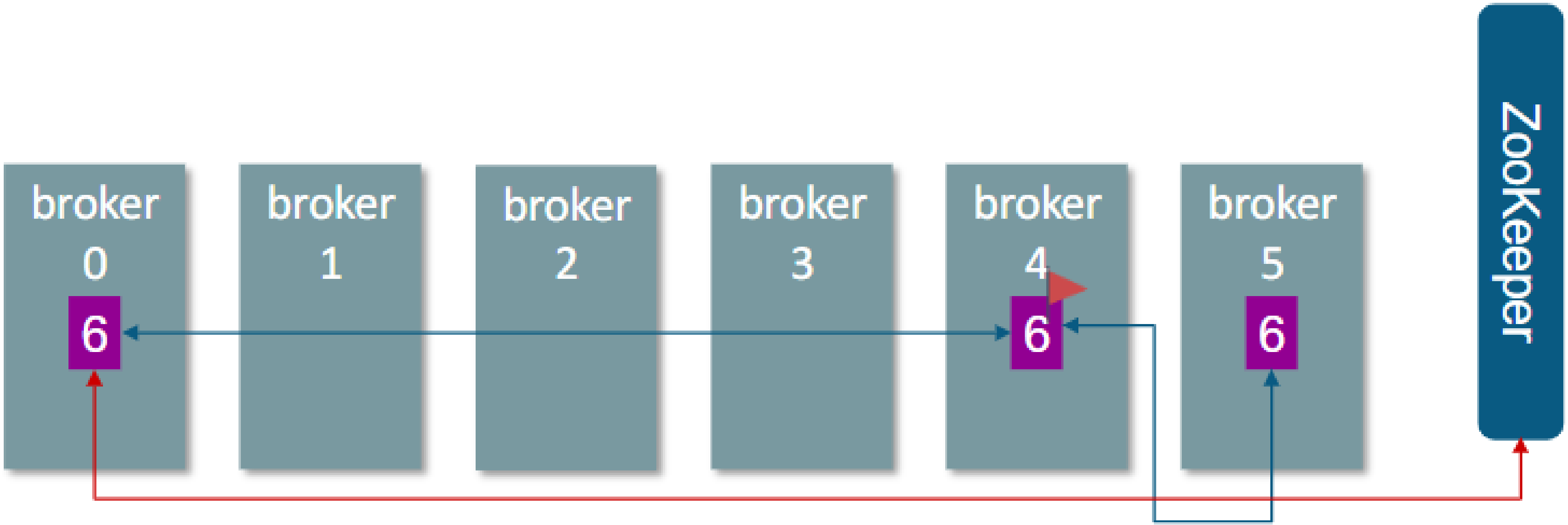
Sending messages as key-value pairs

```
[edureka@localhost kafka_2.12-0.11.0.0]$ bin/kafka-console-producer.sh --broker-
list localhost:9092 --topic my-topic --property "parse.key=true" --property "key
.separator=:"
>key1:value1
>key2:value2
>key3:value3
>
```

Let’s see how Brokers are Configured in a Reliable System

Replication Mechanism

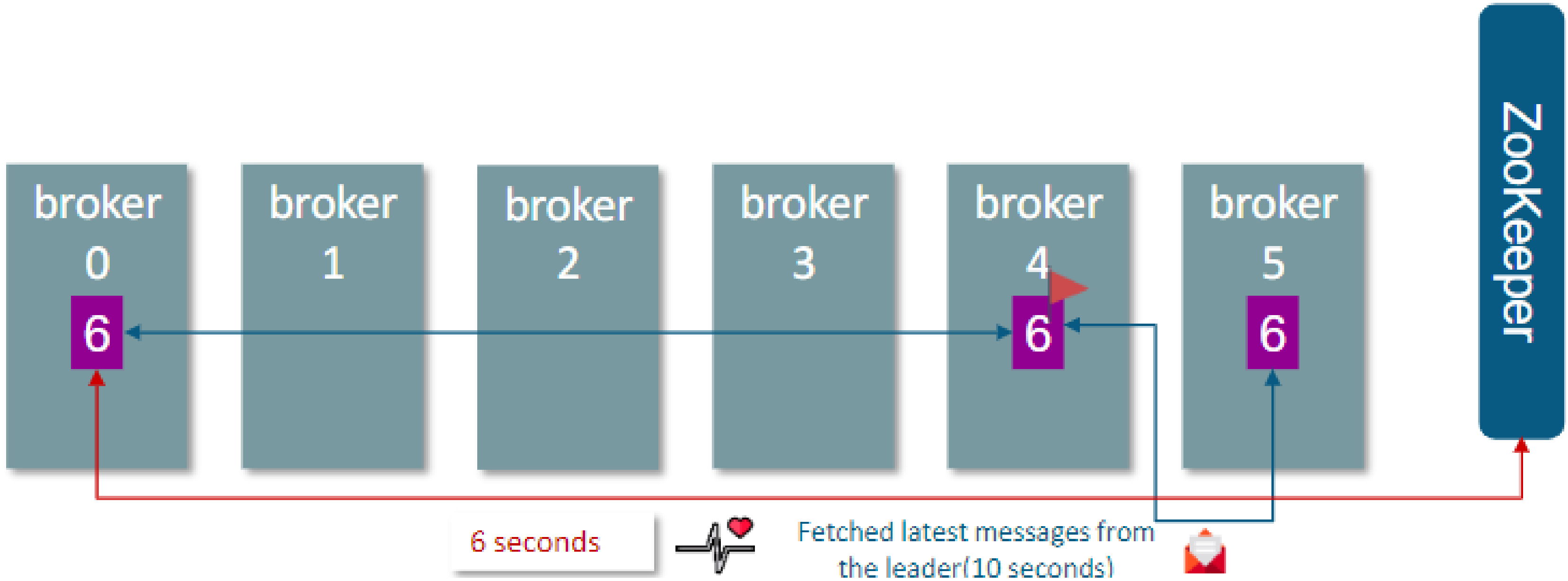
- Kafka's replication mechanism, with its multiple replicas per partition, is at the core of Kafka's reliability guarantees
- Having a message written in multiple replicas is how Kafka provides durability of messages in the event of a crash



Replication Mechanism

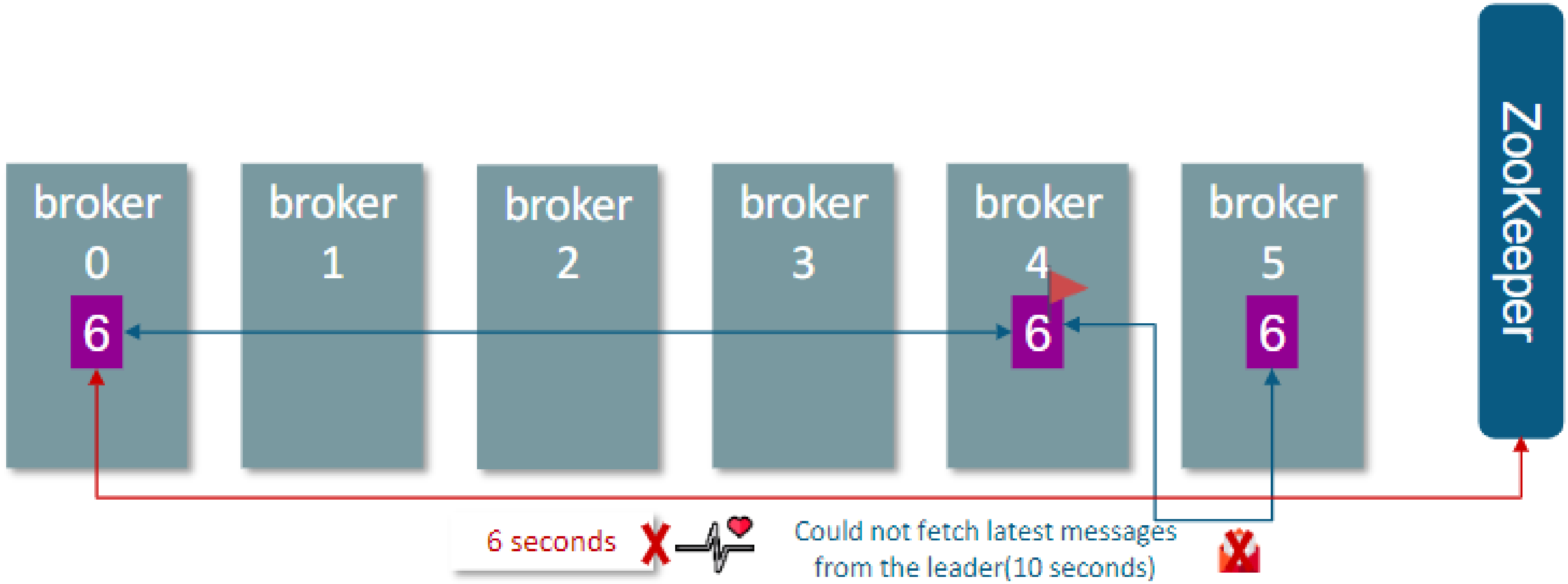
A replica is considered in-sync if it is the leader for a partition, or if it is a follower that:

1. Has an active session with Zookeeper—meaning, it sent a heartbeat to Zookeeper in the last 6 seconds (configurable)
2. Fetched messages from the leader in the last 10 seconds (configurable)



Replication Mechanism

- If a replica loses connection to Zookeeper and can't catch up within 10 seconds, the replica is considered ***out-of-sync***
- An out-of-sync replica gets back into sync when it connects to Zookeeper again and catches up to the most recent message written to the leader
- This usually happens quickly after a temporary network problem is healed



Broker Configuration

cloudera@quickstart:/tmp/kafka-logs/topic-3-0

File Edit View Search Terminal Help

```
[cloudera@quickstart kafka-logs]$ pwd
/tmp/kafka-logs
[cloudera@quickstart kafka-logs]$ cd topic-3-0
[cloudera@quickstart topic-3-0]$ pwd
/tmp/kafka-logs/topic-3-0
[cloudera@quickstart topic-3-0]$ ls
00000000000000000000.index   00000000000000000000.timeindex  leader-epoch-checkpoint
00000000000000000000.log     00000000000000000024.snapshot
[cloudera@quickstart topic-3-0]$ cat 00000000000000000000.log
��������f�/����f�/����������������
                                   MyKey4&Test Java Message 4>
                                                             MyKey7&Test Java Message 7>
                                                                                       MyKey8&Test Java Mes
sage 8>
        MyKey9&Test Java Message 9���N�?����f�1
                                              M�f�1
                                                   R��������������
                                                                  MyKey4&Test Java Message 4>
```

Data preset in the topic is stored in the log files of the respective brokers in the form of byte-array

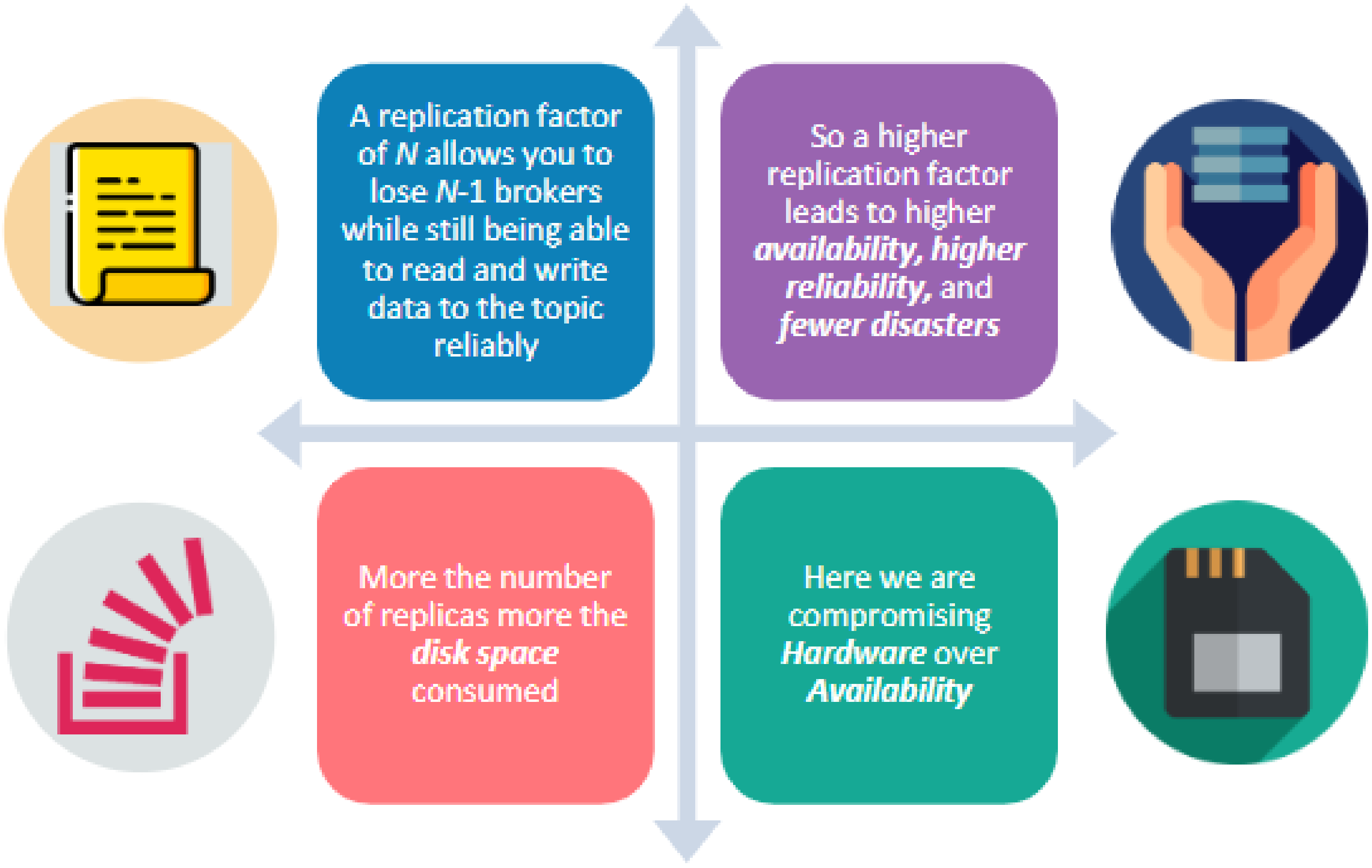
cloudera@quickstart:~/kafka_2.11-0.11.0.1

File Edit View Search Terminal Help

```
[cloudera@quickstart kafka_2.11-0.11.0.1]$ bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic topic-3
Topic:topic-3   PartitionCount:3        ReplicationFactor:3     Configs:
        Topic: topic-3  Partition: 0    Leader: 2       Replicas: 1,0,2 Isr: 2
        Topic: topic-3  Partition: 1    Leader: 2       Replicas: 2,1,0 Isr: 2
        Topic: topic-3  Partition: 2    Leader: 2       Replicas: 0,2,1 Isr: 2
[cloudera@quickstart kafka_2.11-0.11.0.1]$
```

Broker Configurations

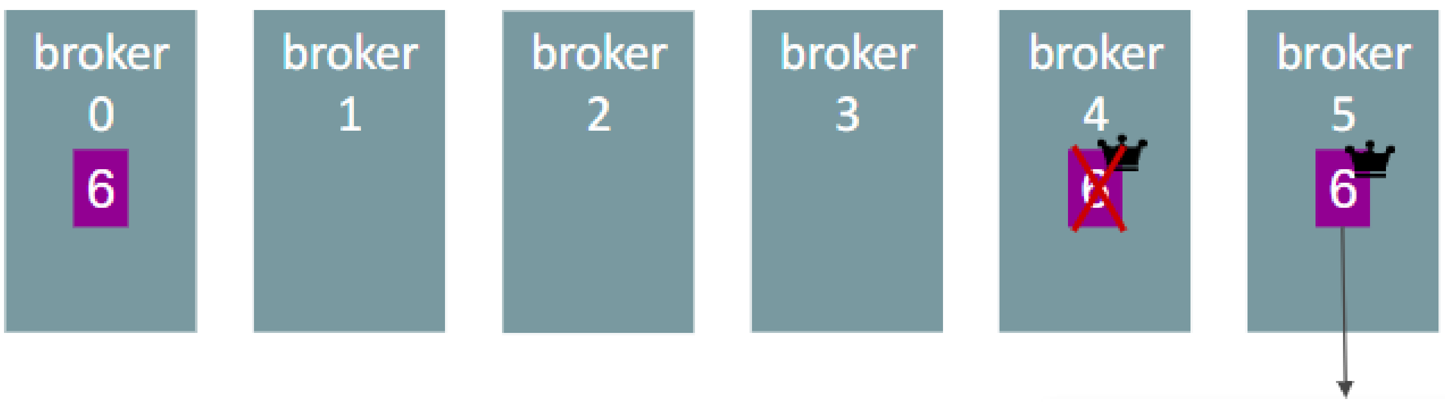
Replication Factor



Let’s see how we can
Elect an Unclean New Leader

Unclean Leader Election

This configuration is only available at the ***broker level***, where parameter ***unclean.leader.election.enable*** comes into picture

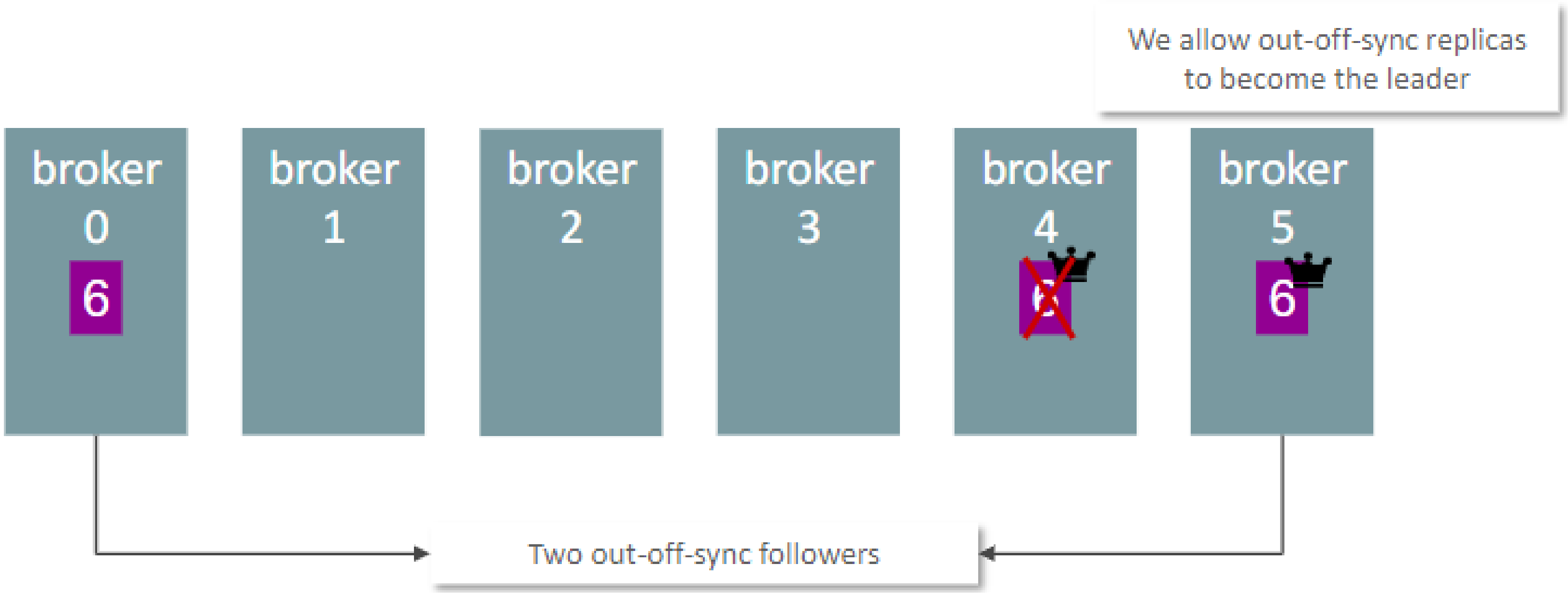


When the leader is not available the in-sync follower replica will be chosen as the new leader

What do we do when no In-Sync-Replica exists except the leader that just became unavailable?

Unclean Leader Election

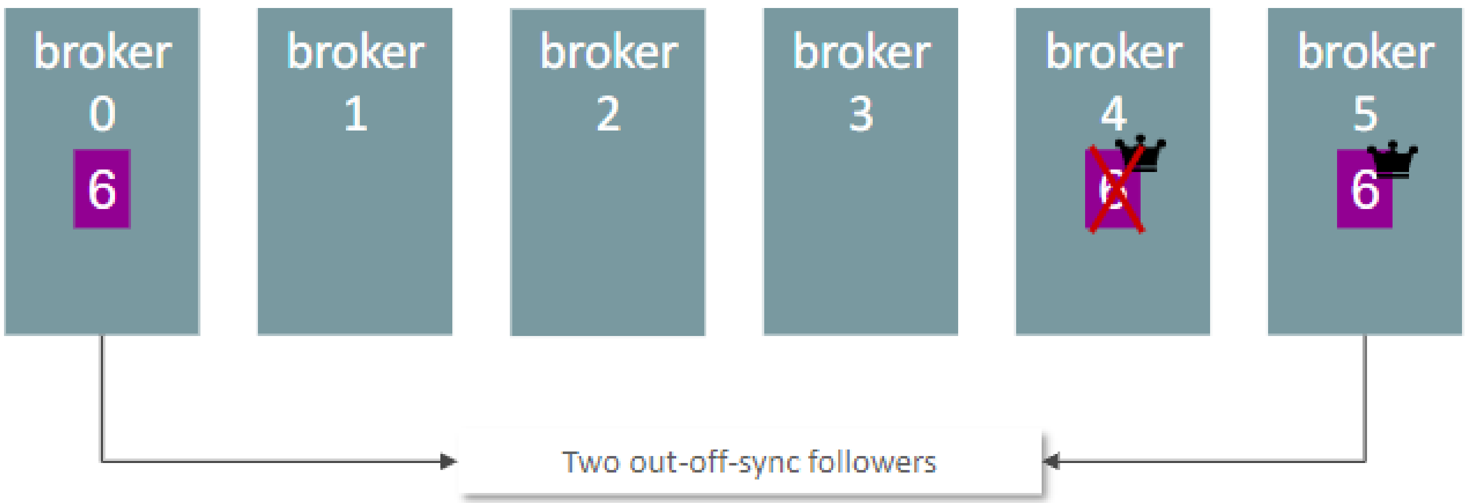
The parameter name is ***unclean.leader.election.enable*** and by default it is set to ***false*** : We allow out-of-sync replicas to become leaders which is called ***unclean leader election***



Unclean Leader Election

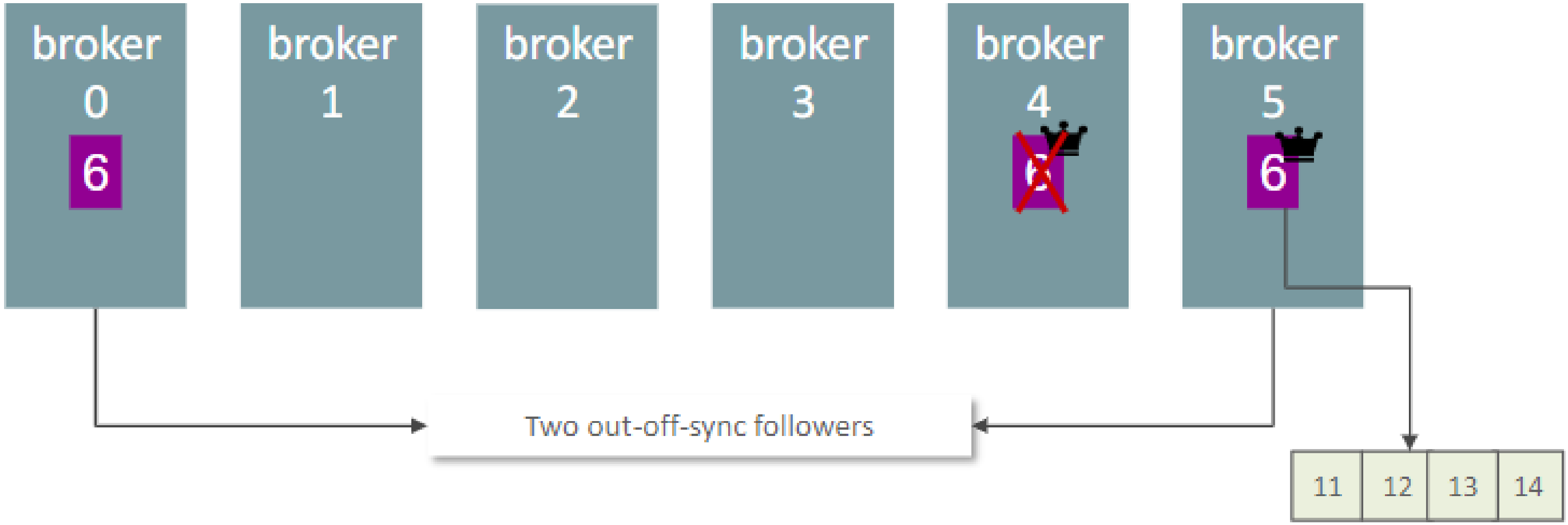
In this scenario, if one of the out-of-sync followers starts first, we have an out-of-sync replica as the only available replica for the partition leader

If we don’t allow the out-of-sync replica to become the new leader, the partition will remain offline until we bring the old leader back online



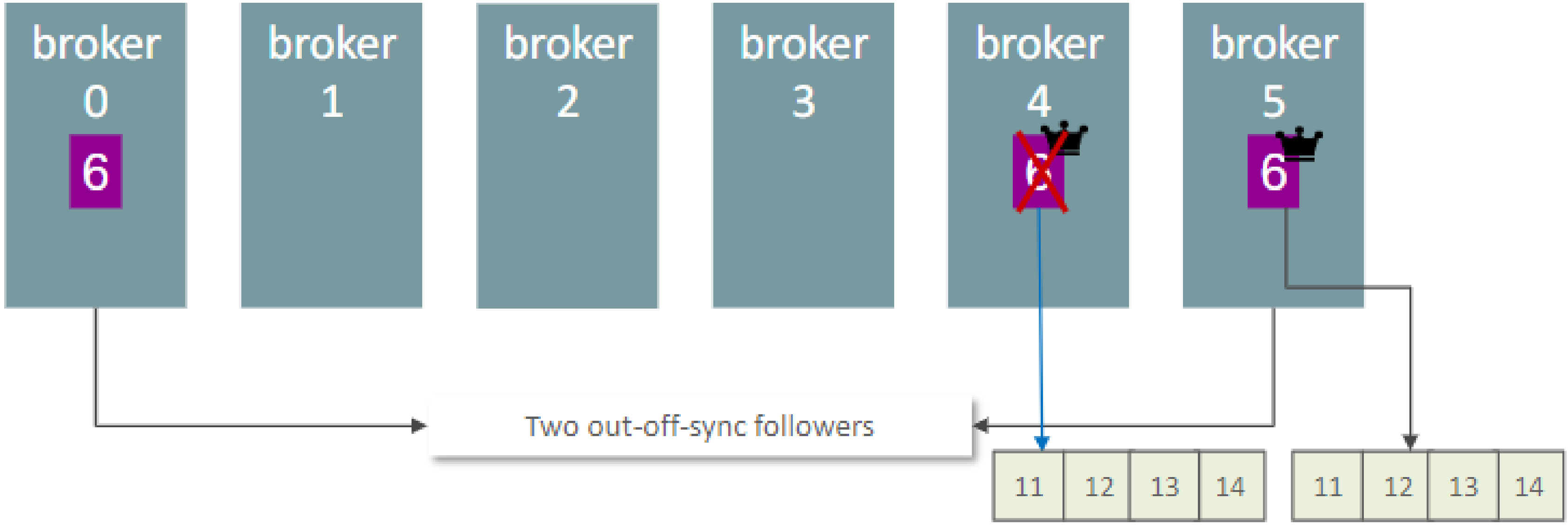
Unclean Leader Election

If we do allow the out-of-sync replica to become the new leader, we are going to lose all messages that were written to the old leader while that replica was out of sync and also cause some inconsistencies in consumers



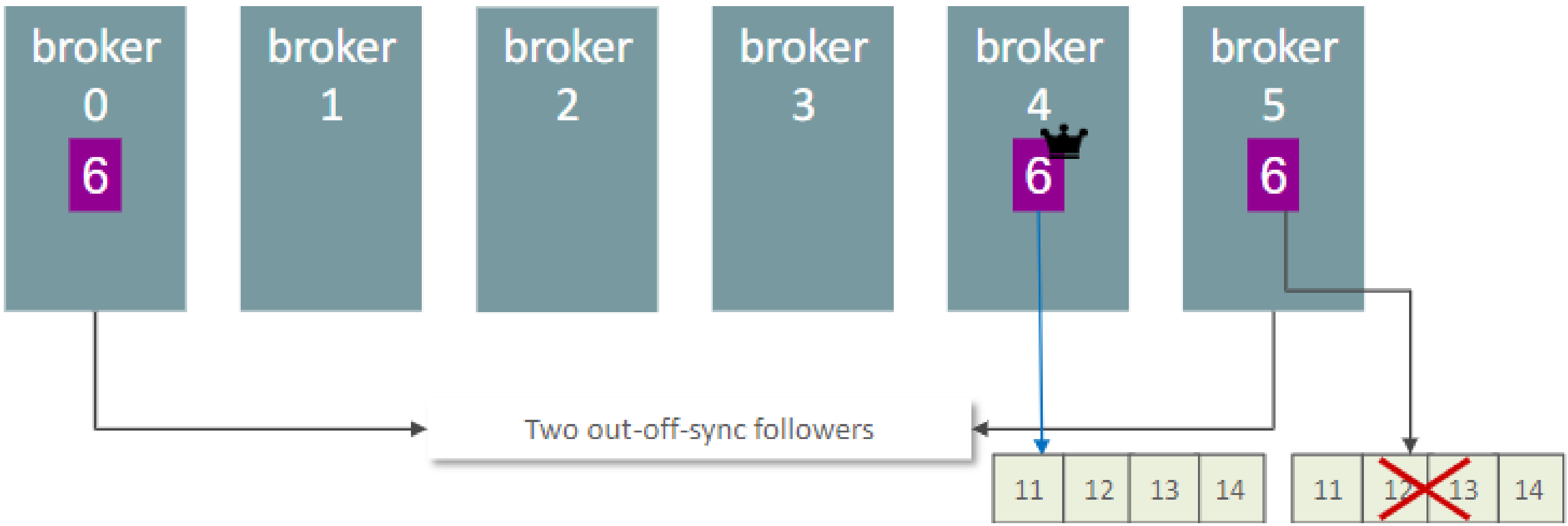
Unclean Leader Election

- If the current leader becomes available, and the old leader is unavailable on broker 4, it will start writing from that offset where it stopped writing, thus having new messages for offset 11 to 14
- Some consumers might have a different set of 11 to 14 messages, while some might have a totally different one



Unclean Leader Election

- In addition, replica on broker 5 will come back online and become a follower of the new leader which is on broker 4
- At that point, it will delete any messages it got that are ahead of the current leader
- Those messages will not be available to any consumer in the future



Let’s see how producers are Configured in a Reliable System

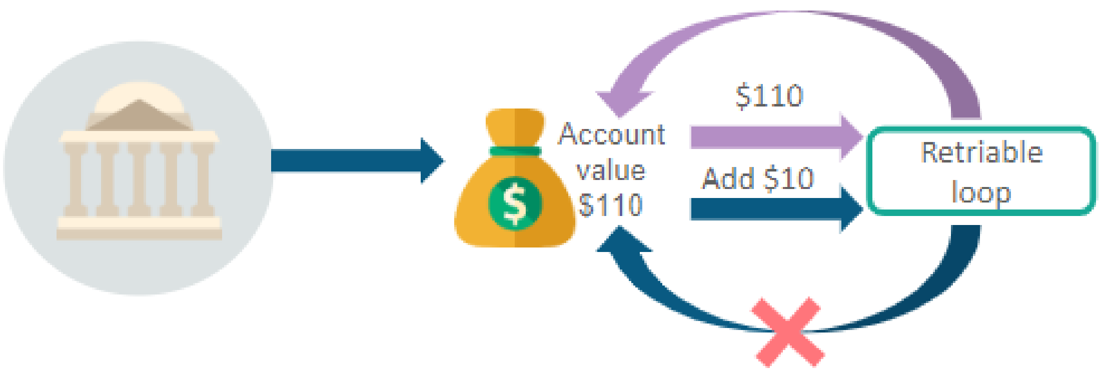
Configuring Producer Retries

Duplicates : Retrying to send a failed message often includes a small risk that both messages were successfully written to the broker, leading to duplicates

Idempotent : Applications make the messages *idempotent* to ensure that even if the same message is sent twice, it has no negative impact on correctness

For example: The message "Account value is $110" is idempotent, since sending it several times doesn't change the result

The message "Add $10 to the account" is not idempotent, since it changes the result every time you send it



After Configuring Kafka it is important to perform some validations

Validating the Configuration

Validating System Reliability is easy to test the broker and client configuration in isolation from the application logic, and it is recommended to do so for two reasons:

- It helps to test if the configuration you've chosen can meet your requirement
- It is good exercise to reason through the expected behaviour of the system

Kafka has 2 tools which are present in the `org.apache.kafka.tools` package which can perform these validations :

VerifiableConsumer Class

- It performs the complementary check
- It consumes events and prints out the events it consumed in order
- It also prints information regarding commits and rebalances

VerifiableProducer Class

- It produces a sequence of messages containing numbers from 1 to a value you choose
- You can configure it by setting the right number of acks, retries, and rate at which the messages will be produced

Tests to run – Leader Election

Leader election: What happens if I kill the leader? How long does it take the producer and consumer to start working as usual again?

Tests to run – Controller Election

Controller election: How long does it take the system to resume after a restart of the controller?

Tests to run – Rolling Restart

Rolling restart: Can I restart the brokers one by one without losing any messages?

Tests to run – Unclean Leader Election

Unclean leader election test: What happens when we kill all the replicas for a partition one by one (to make sure each goes out of sync) and then start a broker that was out of sync? What needs to happen in order to resume operations? Is this acceptable?

Let’s see Performance Tuning in Kafka

Performance Tuning in Kafka

- We need to *tune Kafka to improve it's performance*
- A variety of settings in the configuration files help us do this

Tuning Kafka

Performance tuning involves two important metrics

Latency measures : how long it takes to process one event

Throughput measures : how many events arrive within a specific amount of time

A well tuned Kafka system has just enough brokers to handle topic throughput, given the latency required to process information as it is received

When we talk about tuning Kafka, there are few configuration parameters to be considered

The most important configurations to improve performance are the one, which controls the disk flush rate

We will divide these configurations on component basis as given on next slides

Java/JVM Tuning

- Biggest issue: garbage collection
  - » And, most of the time, the only issue
- Goal is to *minimize GC pause times*
  - » Aka "stop-the-world" events – apps are halted until GC finishes

```
$ java -Xms4g -Xmx4g -XX:PermSize=48m -XX:MaxPermSize=48m
       -XX:+UseG1GC
       -XX:MaxGCPauseMillis=20
       -XX:InitiatingHeapOccupancyPercent=35
```

- Large messages can cause longer garbage collection (GC) pauses as brokers allocate large chunks
- Monitor the GC log and the server log
- If long GC pauses cause Kafka to abandon the ZooKeeper session, you may need to configure longer timeout values for `zookeeper.session.timeout.ms`

Tuning Kafka Producer

Most important configurations which needs to be taken care at Producer side are –

Compression

Use the `compression.codec` and `compressed.topics` producer configuration parameters to enable compression

Gzip and Snappy are supported

Batch size

`batch.size`

Measures batch size in total bytes instead of the number of messages

It controls how many bytes of data to collect before sending messages to the Kafka broker

Set this as high as possible, without exceeding available memory

Sync or Async

`linger.ms`

Sets the maximum time to buffer data in asynchronous mode

By default, the producer does not wait. It sends the buffer any time data is available

Increase `linger.ms` for higher latency and higher throughput in your producer

Tuning Kafka Consumer

Adding more consumers to a group can enhance performance
Adding more consumer groups does not affect performance

The important configuration at Consumer side is –

Fetch size

- We can also have multiple Consumers running to fetch maximum data from partitioned topic available on Kafka Brokers

`fetch.message.max.bytes`

Maximum message size a consumer can read
Must be at least as large as `message.max.bytes`
Default value: 1048576 (1 MiB)

Let’s have a look at Kafka Broker Performance Tuning Properties

Tuning Kafka Broker - Properties

- `message.max.bytes`
- `log.segment.bytes`
- `replica.fetch.max.bytes`
- `num.replica.fetchers`
- `num.io.threads`

Maximum message size the broker will accept

Must be smaller than the consumer `fetch.message.max.bytes`, or the consumer cannot consume the message

Default value: 1000000 (1 MB)

Tuning Kafka Broker - Properties

- message.max.bytes
- **log.segment.bytes**
- replica.fetch.max.bytes
- num.replica.fetchers
- num.io.threads

Size of a Kafka data file. Must be larger than any single message

Tuning Kafka Broker - Properties

- `message.max.bytes`
- `log.segment.bytes`
- **replica.fetch.max.bytes**
- `num.replica.fetchers`
- `num.io.threads`

Maximum message size a broker can replicate
Must be larger than `message.max.bytes`, or a broker can accept messages it cannot replicate, potentially resulting in data loss

Tuning Kafka Broker - Properties

- message.max.bytes
- log.segment.bytes
- replica.fetch.max.bytes
- **num.replica.fetchers**
- num.io.threads

The number of threads which will be replicating data from leader to the follower
If we have threads available we should have more number of replica fetchers to complete replication in parallel

Tuning Kafka Broker - Properties

- message.max.bytes
- log.segment.bytes
- replica.fetch.max.bytes
- num.replica.fetchers
- **num.io.threads**

Setting value for I/O threads directly depends on how much disk you have in your cluster
These threads are used by server for executing request. We should have at least as many threads as we have disks

TEKCRUX
We ReDefine IT