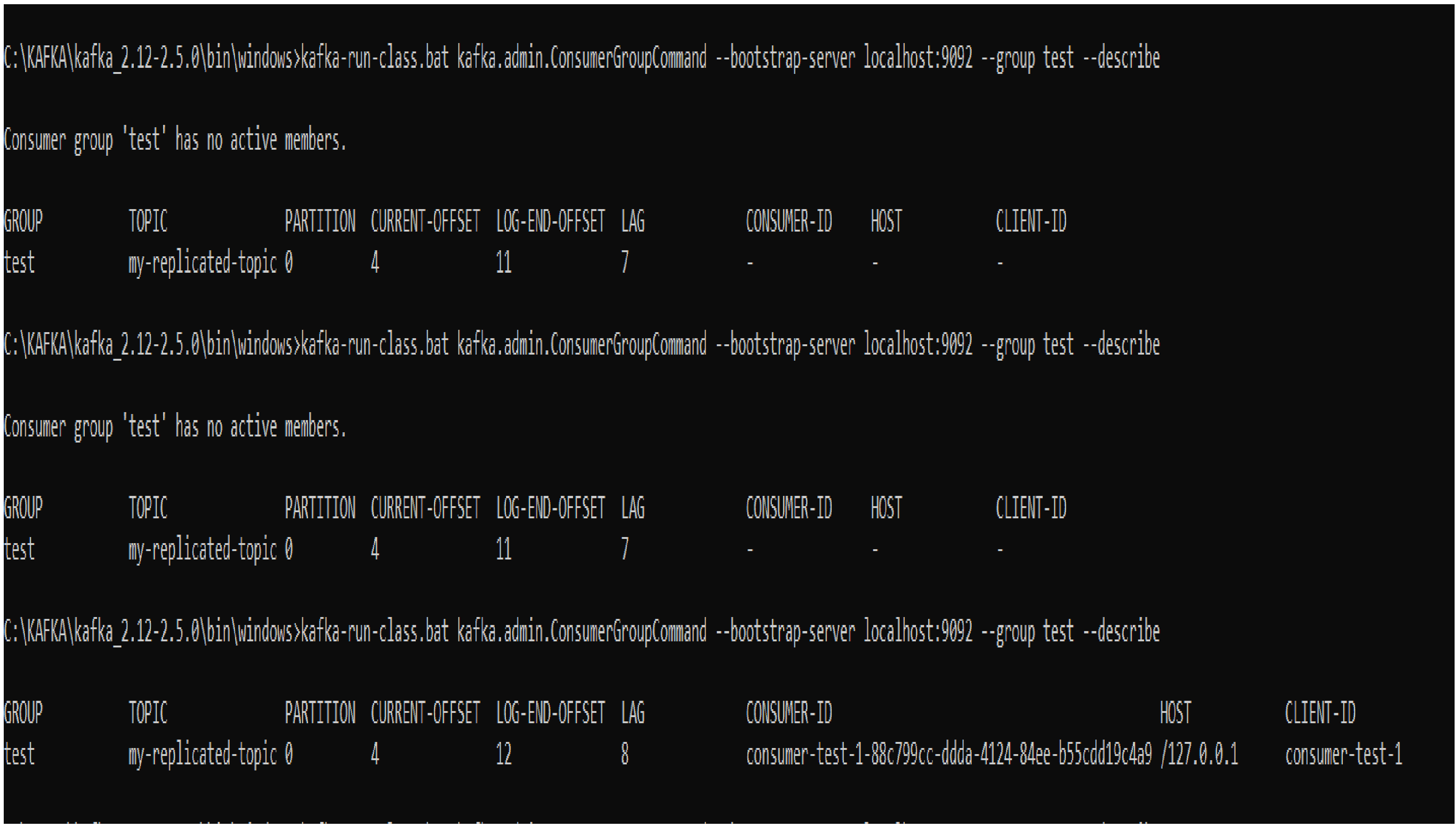
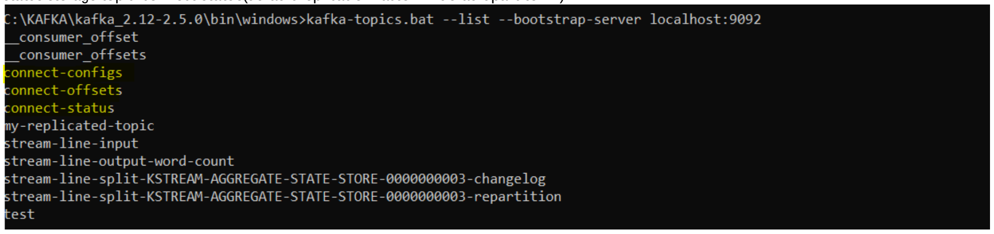


```
C:\KAFKA\kafka_2.12-2.5.0\bin\windows>kafka-run-class.bat kafka.admin.ConsumerGroupCommand --bootstrap-server localhost:9092 --group test --describe

Consumer group 'test' has no active members.

GROUP           TOPIC              PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             CONSUMER-ID     HOST            CLIENT-ID
test            my-replicated-topic 0          4               11              7               -               -               -

C:\KAFKA\kafka_2.12-2.5.0\bin\windows>kafka-run-class.bat kafka.admin.ConsumerGroupCommand --bootstrap-server localhost:9092 --group test --describe

Consumer group 'test' has no active members.

GROUP           TOPIC              PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             CONSUMER-ID     HOST            CLIENT-ID
test            my-replicated-topic 0          4               11              7               -               -               -

C:\KAFKA\kafka_2.12-2.5.0\bin\windows>kafka-run-class.bat kafka.admin.ConsumerGroupCommand --bootstrap-server localhost:9092 --group test --describe

GROUP           TOPIC              PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             CONSUMER-ID                                              HOST            CLIENT-ID
test            my-replicated-topic 0          4               12              8               consumer-test-1-88c799cc-ddda-4124-84ee-b55cdd19c4a9 /127.0.0.1      consumer-test-1
```

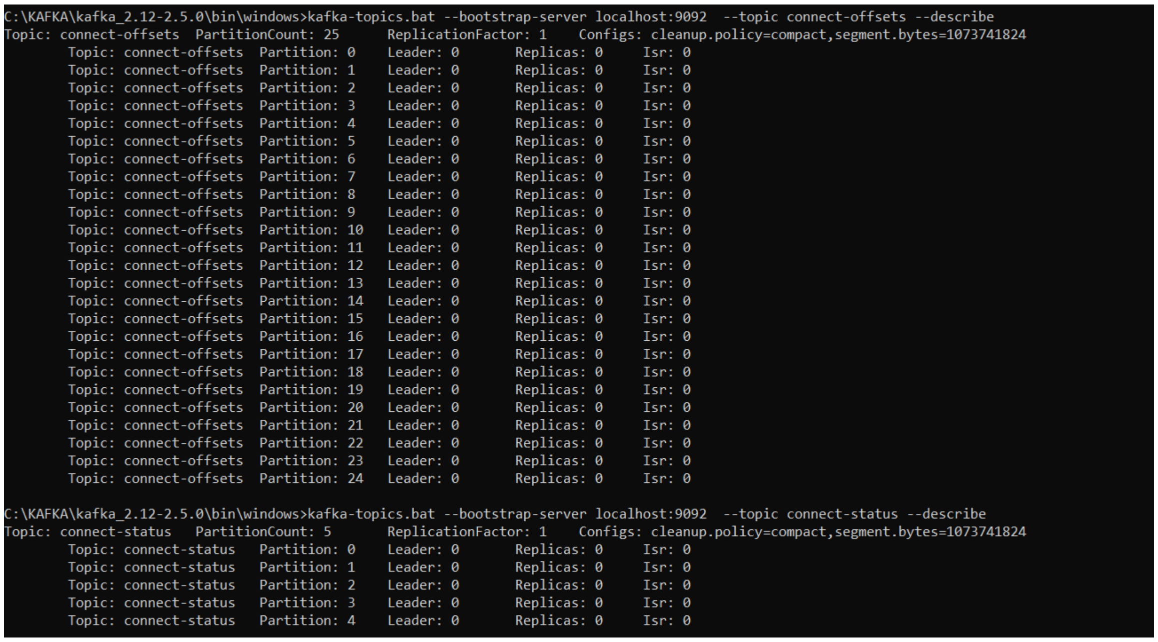
Creating distributed topic

.\bin\windows\connect-distributed.bat .\config\connect-distributed.properties

It create a distributed topic.
The distributed topic internally created 3 other topics (details are present in connect-distributed.properties)
offset.storage.topic=connect-offsets (default replication factor : 1 default partiton: 25)
config.storage.topic=connect-configs(default replication factor :1 default partiton)
status.storage.topic=connect-status(default replication factor : 1 default partiton:1)



```
C:\KAFKA\kafka_2.12-2.5.0\bin\windows>kafka-topics.bat --list --bootstrap-server localhost:9092
__consumer_offset
__consumer_offsets
connect-configs
connect-offsets
connect-status
my-replicated-topic
stream-line-input
stream-line-output-word-count
stream-line-split-KSTREAM-AGGREGATE-STATE-STORE-0000000003-changelog
stream-line-split-KSTREAM-AGGREGATE-STATE-STORE-0000000003-repartition
test
```



```
C:\KAFKA\kafka_2.12-2.5.0\bin\windows>kafka-topics.bat --bootstrap-server localhost:9092  --topic connect-offsets --describe
Topic: connect-offsets  PartitionCount: 25      ReplicationFactor: 1    Configs: cleanup.policy=compact,segment.bytes=1073741824
        Topic: connect-offsets  Partition: 0    Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-offsets  Partition: 1    Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-offsets  Partition: 2    Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-offsets  Partition: 3    Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-offsets  Partition: 4    Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-offsets  Partition: 5    Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-offsets  Partition: 6    Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-offsets  Partition: 7    Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-offsets  Partition: 8    Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-offsets  Partition: 9    Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-offsets  Partition: 10   Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-offsets  Partition: 11   Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-offsets  Partition: 12   Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-offsets  Partition: 13   Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-offsets  Partition: 14   Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-offsets  Partition: 15   Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-offsets  Partition: 16   Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-offsets  Partition: 17   Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-offsets  Partition: 18   Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-offsets  Partition: 19   Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-offsets  Partition: 20   Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-offsets  Partition: 21   Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-offsets  Partition: 22   Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-offsets  Partition: 23   Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-offsets  Partition: 24   Leader: 0       Replicas: 0     Isr: 0

C:\KAFKA\kafka_2.12-2.5.0\bin\windows>kafka-topics.bat --bootstrap-server localhost:9092  --topic connect-status --describe
Topic: connect-status   PartitionCount: 5       ReplicationFactor: 1    Configs: cleanup.policy=compact,segment.bytes=1073741824
        Topic: connect-status   Partition: 0    Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-status   Partition: 1    Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-status   Partition: 2    Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-status   Partition: 3    Leader: 0       Replicas: 0     Isr: 0
        Topic: connect-status   Partition: 4    Leader: 0       Replicas: 0     Isr: 0
```

Rest Endpoint (https://kafka.apache.org/documentation.html#connect)

Since Kafka Connect is intended to be run as a service, it also provides a REST API for managing connectors. The REST API server can be configured using the listeners configuration option. This field should contain a list of listeners in the following format: protocol://host:port,protocol2://host2:port2. Currently supported protocols are http and https. For example:

```
listeners=http://localhost:8080,https://localhost:8443
```

By default, if no listeners are specified, the REST server runs on port 8083 using the HTTP protocol
Since Kafka Connect is intended to be run as a service, it also provides a REST API for managing connectors. The REST API server can be configured using the listeners configuration option. This field should contain a list of listeners in the following format: protocol://host:port,protocol2://host2:port2. Currently supported protocols are http and https. For example:

```
listeners=http://localhost:8080,https://localhost:8443
```

By default, if no listeners are specified, the REST server runs on port 8083 using the HTTP protocol.

When using HTTPS, the configuration has to include the SSL configuration. By default, it will use the ssl.* settings. In case it is needed to use different configuration for the REST API than for connecting to Kafka brokers, the fields can be prefixed with listeners.https. When using the prefix, only the prefixed options will be used and the ssl.* options without the prefix will be ignored

The REST API is used not only by users to monitor / manage Kafka Connect. It is also used for the Kafka Connect cross-cluster communication. Requests received on the follower nodes REST API will be forwarded to the leader node REST API. In case the URI under which is given host reachable is different from the URI which it listens on, the configuration options rest.advertised.host.name, rest.advertised.port and rest.advertised.listener can be used to change the URI which will be used by the follower nodes to connect with the leader. When using both HTTP and HTTPS listeners, the rest.advertised.listener option can be also used to define which listener will be used for the cross-cluster communication. When using HTTPS for communication between nodes, the same ssl.* or listeners.https options will be used to configure the HTTPS client.

The following are the currently supported REST API endpoints:

- GET /connectors - return a list of active connectors
- POST /connectors - create a new connector; the request body should be a JSON object containing a string name field and an object config field with the connector configuration parameters
- GET /connectors/{name} - get information about a specific connector
- GET /connectors/{name}/config - get the configuration parameters for a specific connector
- PUT /connectors/{name}/config - update the configuration parameters for a specific connector
- GET /connectors/{name}/status - get current status of the connector, including if it is running, failed, paused, etc., which worker it is assigned to, error information if it has failed, and the state of all its tasks
- GET /connectors/{name}/tasks - get a list of tasks currently running for a connector
- GET /connectors/{name}/tasks/{taskid}/status - get current status of the task, including if it is running, failed, paused, etc., which worker it is assigned to, and error information if it has failed
- PUT /connectors/{name}/pause - pause the connector and its tasks, which stops message processing until the connector is resumed
- PUT /connectors/{name}/resume - resume a paused connector (or do nothing if the connector is not paused)
- POST /connectors/{name}/restart - restart a connector (typically because it has failed)
- POST /connectors/{name}/tasks/{taskId}/restart - restart an individual task (typically because it has failed)
- DELETE /connectors/{name} - delete a connector, halting all tasks and deleting its configuration
- GET /connectors/{name}/topics - get the set of topics that a specific connector is using since the connector was created or since a request to reset its set of active topics was issued
- PUT /connectors/{name}/topics/reset - send a request to empty the set of active topics of a connector

From <https://kafka.apache.org/documentation.html#connect>

Rest to be continuedd…….

Updating Broker Configs

Type of broker configs

- read-only: Requires a broker restart for update
- per-broker: May be updated dynamically for each broker
- cluster-wide: May be updated dynamically as a cluster-wide default. May also be updated as a per-broker value for testing.

To alter the current broker configs for broker id 0 (for example, the number of log cleaner threads):

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type brokers --entity-name 0 --alter --add-config log.cleaner.threads=2
```

To describe the current dynamic broker configs for broker id 0:

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type brokers --entity-name 0 --describe
```

To delete a config override and revert to the statically configured or default value for broker id 0 (for example, the number of log cleaner threads):

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type brokers --entity-name 0 --alter --delete-config log.cleaner.threads
```

Some configs may be configured as a cluster-wide default to maintain consistent values across the whole cluster. All brokers in the cluster will process the cluster default update.
For example, to update log cleaner threads on all brokers:

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type brokers --entity-default --alter --add-config log.cleaner.threads=2
```

To describe the currently configured dynamic cluster-wide default configs:

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type brokers --entity-default --describe
```

All configs that are configurable at cluster level may also be configured at per-broker level (e.g. for testing). If a config value is defined at different levels, the following order of precedence is used:

- Dynamic per-broker config stored in ZooKeeper
- Dynamic cluster-wide default config stored in ZooKeeper
- Static broker config from server.properties
- Kafka default

End date 21/7/2020
Start from
Constant Time Suffices

From <https://kafka.apache.org/documentation/#producerapi>

How kafka dealt with Efficieny issues :-

We have put significant effort into efficiency. One of our primary use cases is handling web activity data, which is very high volume: each page view may generate dozens of writes. Furthermore, we assume each message published is read by at least one consumer (often many), hence we strive to make consumption as cheap as possible.
We have also found, from experience building and running a number of similar systems, that efficiency is a key to effective multi-tenant operations. If the downstream infrastructure service can easily become a bottleneck due to a small bump in usage by the application, such small changes will often create problems. By being very fast we help ensure that the application will tip-over under load before the infrastructure. This is particularly important when trying to run a centralized service that supports dozens or hundreds of applications on a centralized cluster as changes in usage patterns are a near-daily occurrence.
We discussed disk efficiency in the previous section. Once poor disk access patterns have been eliminated, there are two common causes of inefficiency in this type of system: too many small I/O operations, and excessive byte copying.
The small I/O problem happens both between the client and the server and in the server's own persistent operations.
To avoid this, our protocol is built around a "message set" abstraction that naturally groups messages together. This allows network requests to group messages together and amortize the overhead of the network roundtrip rather than sending a single message at a time. The server in turn appends chunks of messages to its log in one go, and the consumer fetches large linear chunks at a time.
This simple optimization produces orders of magnitude speed up. Batching leads to larger network packets, larger sequential disk operations, contiguous memory blocks, and so on, all of which allows Kafka to turn a bursty stream of random message writes into linear writes that flow to the consumers.
The other inefficiency is in byte copying. At low message rates this is not an issue, but under load the impact is significant. To avoid this we employ a standardized binary message format that is shared by the producer, the broker, and the consumer (so data chunks can be transferred without modification between them).
The message log maintained by the broker is itself just a directory of files, each populated by a sequence of message sets that have been written to disk in the same format used by the producer and consumer. Maintaining this common format allows optimization of the most important operation: network transfer of persistent log chunks. Modern unix operating systems offer a highly optimized code path for transferring data out of pagecache to a socket; in Linux this is done with the sendfile system call.
To understand the impact of sendfile, it is important to understand the common data path for transfer of data from file to socket:

1. **The operating system reads data from the disk into pagecache in kernel space**
2. **The application reads the data from kernel space into a user-space buffer**
3. **The application writes the data back into kernel space into a socket buffer**
4. **The operating system copies the data from the socket buffer to the NIC buffer where it is sent over the network**

This is clearly inefficient, there are four copies and two system calls. Using sendfile, this re-copying is avoided by allowing the OS to send the data from pagecache to the network directly. So in this optimized path, only the final copy to the NIC buffer is needed.
We expect a common use case to be multiple consumers on a topic. Using the zero-copy optimization above, data is copied into pagecache exactly once and reused on each consumption instead of being stored in memory and copied out to user-space every time it is read. This allows messages to be consumed at a rate that approaches the limit of the network connection.
This combination of pagecache and sendfile means that on a Kafka cluster where the consumers are mostly caught up you will see no read activity on the disks whatsoever as they will be serving data entirely from cache.
For more background on the sendfile and **zero-copy support** in Java, see this article.

Interesting read about ZERO Copy at https://developer.ibm.com/articles/j-zerocopy/#download >> Really Interesting

From <https://kafka.apache.org/documentation/#producerapi>

KIP-345: Introduce static membership protocol to reduce consumer rebalances

From <https://cwiki.apache.org/confluence/display/KAFKA/KIP-345%3A+Introduce+static+membership+protocol+to+reduce+consumer+rebalances>

STATIC member ship protocol :-

Replicated Logs: Quorums, ISRs, and State Machines (Oh my!)

https://kafka.apache.org/documentation/#design_replicatedlog
A replicated log models the process of coming into consensus on the order of a series of values (generally numbering the log entries 0, 1, 2, ...). There are many ways to implement this, but the simplest and fastest is with a leader who chooses the ordering of values provided to it. As long as the leader remains alive, all followers need to only copy the values and ordering the leader chooses.

Of course if leaders didn't fail we wouldn't need followers! When the leader does die we need to choose a new leader from among the followers. But followers themselves may fall behind or crash so we must ensure we choose an up-to-date follower. The fundamental guarantee a log replication algorithm must provide is that if we tell the client a message is committed, and the leader fails, the new leader we elect must also have that message. This yields a tradeoff: if the leader waits for more followers to acknowledge a message before declaring it committed then there will be more potentially electable leaders.

If you choose the number of acknowledgements required and the number of logs that must be compared to elect a leader such that there is guaranteed to be an overlap, then this is called a Quorum.

From <https://kafka.apache.org/documentation/#design_replicatedlog>

Kafka takes a slightly different approach to choosing its quorum set. Instead of majority vote, Kafka dynamically maintains a set of in-sync replicas (ISR) that are caught-up to the leader. Only members of this set are eligible for election as leader. A write to a Kafka partition is not considered committed until *all* in-sync replicas have received the write. This ISR set is persisted to ZooKeeper whenever it changes. Because of this, any replica in the ISR is eligible to be elected leader. This is an important factor for Kafka's usage model where there are many partitions and ensuring leadership balance is important. With this ISR model and *f+ 1* replicas, a Kafka topic can tolerate *f* failures without losing committed messages.

Unclean leader election: What if they all die?

From <https://kafka.apache.org/documentation/#design_replicatedlog>

Note that Kafka's guarantee with respect to data loss is predicated on at least one replica remaining in sync. If all the nodes replicating a partition die, this guarantee no longer holds.
However a practical system needs to do something reasonable when all the replicas die. If you are unlucky enough to have this occur, it is important to consider what will happen. There are two behaviors that could be implemented:

1. Wait for a replica in the ISR to come back to life and choose this replica as the leader (hopefully it still has all its data).
2. Choose the first replica (not necessarily in the ISR) that comes back to life as the leader.

By default from version 0.11.0.0, Kafka chooses the first strategy and favor waiting for a consistent replica. This behavior can be changed using configuration property unclean.leader.election.enable, to support use cases where uptime is preferable to consistency.

Availability and Durability Guarantees

From <https://kafka.apache.org/documentation/#design_replicatedlog>
we provide two topic-level configurations that can be used to prefer message durability over availability:

1. **Disable unclean leader election** - if all replicas become unavailable, then the **partition will remain unavailable until the most recent leader becomes available again**. **This effectively prefers unavailability over the risk of message loss**. See the previous section on Unclean Leader Election for clarification.
2. **Specify a minimum ISR size** - **the partition will only accept writes if the size of the ISR is above a certain minimum**, in order to prevent the loss of messages that were written to just a single replica, which subsequently becomes unavailable. This setting only takes effect if the producer uses acks=all and guarantees that the message will be acknowledged by at least this many in-sync replicas. This setting offers a trade-off between consistency and availability. A higher setting for minimum ISR size guarantees better consistency since the message is guaranteed to be written to more replicas which reduces the probability that it will be lost. However, it reduces availability since the partition will be unavailable for writes if the number of in-sync replicas drops below the minimum threshold.

Replica Management

From <https://kafka.apache.org/documentation/#design_replicatedlog>

Broker Node Registry

From <https://kafka.apache.org/documentation/#design_replicatedlog>
On startup, **a broker node registers itself by creating a znode with the logical broker id under /brokers/ids**. The **purpose of the logical broker id is to allow a broker to be moved to a different physical machine without affecting consumers**. An attempt to register a broker id that is already in use (say because two servers are configured with the same broker id) results in an error.
Since the broker registers itself in ZooKeeper using ephemeral znodes, this registration is dynamic and will disappear if the broker is shutdown or dies (thus notifying consumers it is no longer available).

Broker Topic Registry

From <https://kafka.apache.org/documentation/#design_replicatedlog>
Each broker registers itself under the topics it maintains and stores the number of partitions for that topic.

Cluster Id

T**he cluster id is a unique and immutable identifier assigned to a Kafka cluster**. **The cluster id can have a maximum of 22 characters and the allowed characters are defined by the regular expression [a-zA-Z0-9_\-]+**, which corresponds to the characters used by the URL-safe Base64 variant with no padding. Conceptually, it is auto-generated when a cluster is started for the first time.
Implementation-wise, **it is generated when a broker with version 0.10.1 or later is successfully started for the first time**. **The broker tries to get the cluster id from the /cluster/id znode during startup. If the znode does not exist, the broker generates a new cluster id and creates the znode with this cluster id.**

From <https://kafka.apache.org/documentation/#design_replicatedlog>

Broker node registration

The broker nodes are basically independent, so they only publish information about what they have. When a broker joins, it registers itself under the broker node registry directory and writes information about its host name and port. The broker also register the list of existing topics and their logical partitions in the broker topic registry. New topics are registered dynamically when they are created on the broker.

From <https://kafka.apache.org/documentation/#design_replicatedlog>