KIP-966: Eligible Leader Replicas

Status

Current state: *Accepted*

Discussion thread: *here*

Vote thread: *here*

JIRA: *here*

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

A partition replica can experience local data loss in unclean shutdown scenarios where unflushed data in the OS page cache is lost - such as an availability zone power outage or a server error. The Kafka replication protocol is designed to handle these situations by removing such replicas from the ISR and only re-adding them once they have caught up and therefore recovered any lost data. This prevents replicas that lost an arbitrary log suffix, which included committed data, from being elected leader.

However, there is a "last replica standing" state which when combined with a data loss unclean shutdown event can turn a local data loss scenario into a global data loss scenario, i.e., committed data can be removed from all replicas. When the last replica in the ISR experiences an unclean shutdown and loses committed data, it will be reelected leader after starting up again, causing rejoining followers to truncate their logs and thereby removing the last copies of the committed records which the leader lost initially.

This proposal solves this "last replica standing" data loss issue in KRaft clusters, providing MinISR-1 tolerance to data loss unclean shutdown events.

Consider the following "last replica standing" scenario with a partition with 3 healthy replicas in the ISR(0,1,2) and the min.ISR is 2.

- At T0, a network partitioning happens and broker 0 gets out of ISR.
- At T1, another network partitioning happens, and broker 1 also leaves. Broker 2 becomes the leader of this partition.
- At T2, broker 2 suffers an unclean shutdown which also causes broker 2 to lose some of its logs. The current kafka behavior will prevent ISR drops to empty which keeps the last replica broker 2. Also, it puts this partition to no leader state.
- At T3, the network partitioning is done. Broker 0 and broker 1 come back. However, the ISR can't be recovered because these two brokers are not in ISR.
- At T4, broker 2 restarts and becomes the leader. Then, the replication begins and results in global data loss.

Proposed Changes

With ZK marked deprecated in AK 3.5, only the fix in KRaft is in scope.

Additional High Watermark advancement requirement

We propose to enforce that **High Watermark can only advance if the ISR size is larger or equal to *min.insync.replicas***.

To help you understand how we came up with this proposal. A quick recap of some key concepts.

- High Watermark.
    - In ISR, each server maintains a high watermark, which represents the highest offset of the replicated log known to be committed / durably stored.
    - Also, for consumers, only the message above the High Watermark is visible to them.
- Ack=0/1/all produce request. It defines when the Kafka server should respond to the produce request.
    - For ack=0 requests, the server will not respond to the message and just put it into the leader's local log.
    - For ack=1 requests, the server should respond when the message is persisted in the leader's local log.
    - For ack=all requests, the server should respond when the message is persisted in all the ISR members' local log and the size of the ISR member is larger than min ISR.

In order to have a message sent to the server being available for a consumer to fetch, there are 2 main steps. First, the produce request has to be acked by the server. This can be controlled by the produce request's ack requirement. Second, the replication is good enough to advance the HWM above the message. Notably, in the current system, min ISR is a factor only useful when accepting an ack=all request. It has no use in accepting ack=0/1 requests and the HWM advancement. This behavior complicates the durability model in a mixed type of requests workload.

We mostly want to enhance the durability of the ack=all requests. In the scenario raised in the motivation section, the server may receive both ack=0/1 and ack=all messages during T1. During this period, the server will reject the ack=all requests due to not enough replicas to meet the min ISR, but it accepts the ack=0/1 requests. Also, because the leader is the only one in the ISR, it is allowed to advance the HWM to the end of its log. Let's say there is some extra info somewhere to let the controller choose broker 1 as the leader at T4 instead. Then there is no data loss for the ack=all requests, but the HWM stored on broker 1 is lower than broker 2. This can cause HWM to move backward and impacts the consumers.

To avoid the ack=0/1 message interference, we propose to have the above additional requirement on the HWM advancement. Here are some clarifications:

- It applies to the ack=0/1 message replication as well. Note that the leader still acknowledges the client requests when the ack=1 messages have persisted in the leader log.
- The ISR membership refers to the latest ISR membership persisted by the controller, not the "maximal ISR" which is defined by the leader that includes the current ISR members and pending-to-add replicas that have not yet been committed to the controller.
- If maximal ISR > ISR, the message should be replicated to the maximal ISR before covering the message under HWM. The ISR refers to the ISR committed by the controller.

As a side effect of the new requirement:

- The current Kafka cluster does allow the following topic creation configs:
    - *min.insync.replicas* > replication factor
    - *min.insync.replicas* > current cluster size

With the proposal, the ack=0/1 requests will all be acknowledged by the server with the above config, however, no messages can be visible to the clients. For backward compatibility, the effective *min.insync.replicas* will be min(*min.insync.replicas, replication factor*).

Eligible Leader Replicas

Our ultimate goal is to elect a leader without data loss or HWM moving backward when we only have min ISR - 1 unclean shutdowns. Before we introduce the new mechanism, let's recap the ISR.

The current ISR primarily serves two purposes.

1. It acts as a quorum for replication. The High Watermark is utilized to indicate the lower bound of the log offset that all ISR replicas have replicated.
2. It functions as a candidate set for the leader. In the case of produce requests with ack=all, the leader will commit to the request only when the message has been replicated to the entire ISR. Thus, the controller ensures data safety by selecting any broker within the ISR as the leader.

Typically, the second function can be inferred from the first one. However, the current Kafka server promises that the server will commit to ack=all messages only if the ISR size is at least *min.insync.replicas*. Also, as we are adding the new HWM requirement to avoid the HWM moving backward, this set of "min ISR" rules establishes ISR as a sufficient but not necessary condition for leader election. Consequently, in the background scenario, we can ensure the durability of the ack=all messages if we are somehow able to elect the out-of-ISR member broker 1.

Therefore, we propose to separate the functions of the original ISR.

- The ISR will still continue to serve its replication function. The High Watermark forwarding still requires the replication of the full ISR. This ensures that replication between brokers remains unchanged.
- To handle leader elections, we will introduce a concept called *Eligible Leader Replicas* (ELR). In addition to the ISR members, replicas in ELR are also eligible for leader election during a clean election process.

At a high level, we use ELR to store the replicas that are not in ISR but guarantee to have the data at least to High Watermark.

ISR invariants:

- The ISR can be empty now. The proposal maintains the behavior of removing a replica out of ISR if it is lagging from the ISR or it is fenced by the controller.

ELR invariants:

- The member of ELR should not be in ISR.
- The member of ELR should have the data at least to HWM.
- The member of ELR can lag in replication or in an unknown status from the controller's perspective(fenced).
- If ELR is not empty, the ISR is under min ISR.
- ELR + ISR size will not be dropped below the min ISR unless the controller discovers an ELR member has an unclean shutdown.
    - The controller will remove the ELR member if it registers with an unclean shutdown.
    - The unclean shutdown detection is discussed in another section below.

Broker behaviors:

Both the follower and the leader don’t have any new behavior to handle ELR. They still refer to ISR for decision making.

Controller behaviors:

ELR will be maintained purely on the controller side in the partition state. There are 4 ways to interact with the ELR:

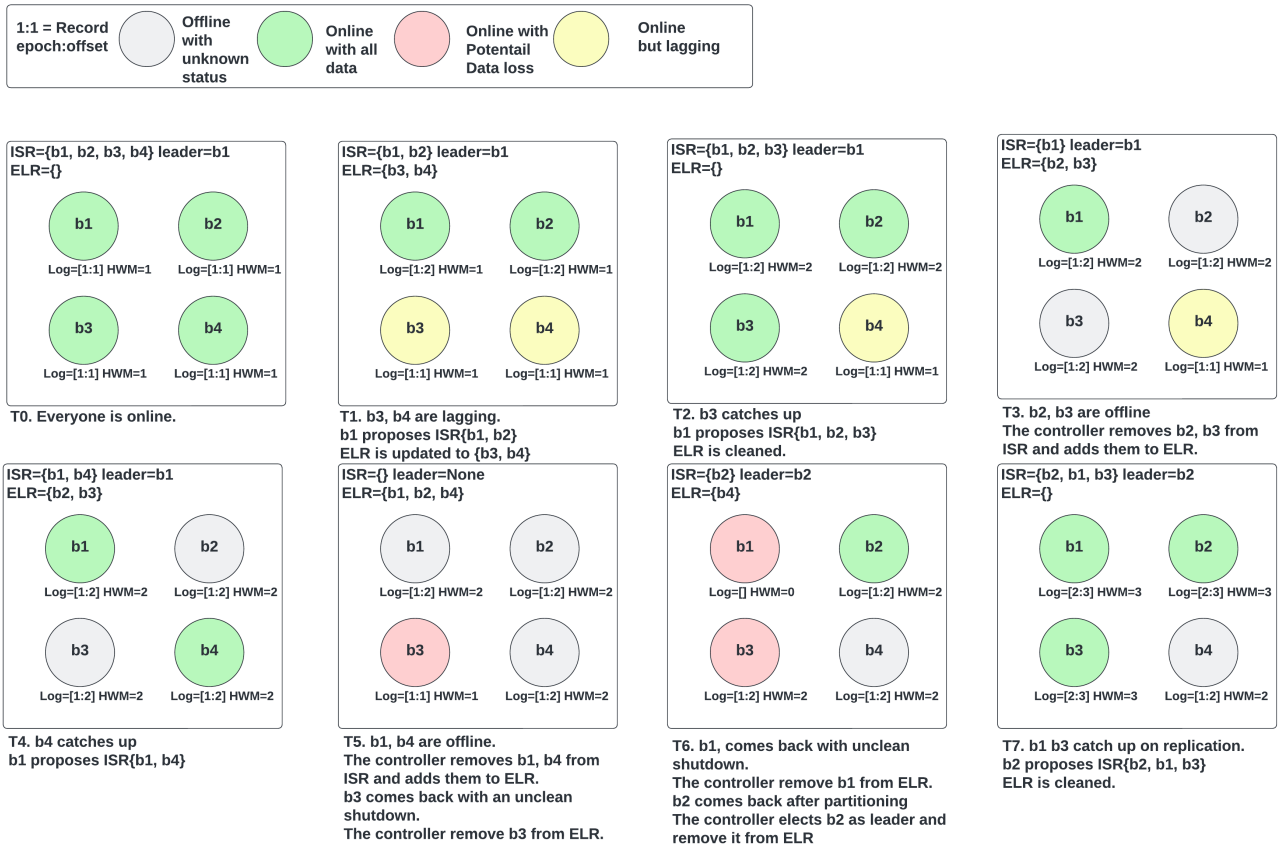
1. *AlterPartition* request. Although brokers are not explicitly aware of the ELR, they can indirectly modify it through the AlterPartition request. When the controller receives the new ISR, it will trigger an ELR update.
2. The replica gets fenced. When it happens, the controller will trigger the ELR update with the new updated ISR.
3. The replica gets unfenced. If the replica is an ELR member and ISR is empty, this replica will be elected as leader, added to ISR, and removed from ELR.
4. During the broker registration, if the broker had an unclean shutdown, the controller will remove the broker from ISR and ELR before persisting the registration record.

ELR update will take a proposed ISR and the controller does the following:

1. When the proposed ISR is larger or equal to min ISR, the controller will update the ISR and empty the ELR.
2. When the proposed ISR is smaller than min ISR, the controller will
    a. retain the current members of the ELR.
    b. add (the current ISR - the proposed ISR) to ELR.
    c. remove the duplicate member in both ISR and ELR from ELR.

The high-level guide and the reasoning behind the above update rules are that ELR will only exist when the ISR is below min ISR. At this moment, the HWM will not advance. Also, only the member in the last ISR snapshot when the ISR drops below min ISR can join the ELR which indicates the ELR member has the logs at least to the HWM.

Here is an example that demonstrates most of the above ELR behaviors. The setup is 4 brokers with min ISR 3.



A common question is whether we could advance HWM when we have an ELR member (not replicating from leader), thus violating the invariant that every ELR member has data at least up to HWM. Consider the following example of a 3 replicas partition with min ISR=2:

1. ISR=[0], ELR=[1], broker 0 is the leader. Broker 1 is up but doesn't have network connectivity.
2. Broker 2 comes up catches up and the leader wants to add it to ISR. At this point ISR=[0], but Maximal ISR=[0, 2].
3. Currently, we would advance HWM because it replicated to 2 brokers (the ones in Maximal ISR), but in the new protocol we wait until the controller updates ISR=[0,2] to avoid advancing HWM beyond what ELR=[1] has.

Other behaviors:

1. Change of the ELR does not require a leader epoch bump. In most cases, the ELR updates along with the ISR changes. The only case of the ELR changes alone is when an ELR broker registers after an unclean shutdown. In this case, no need to bump the leader epoch.
2. When updating the config *min.insync.replicas,* if the new min ISR <= current ISR, the ELR will be removed.
3. A new metric of Electable leaders will be added. It reflects the count of (ISR + ELR).
4. The *AlterPartitionReassignments*. The leader updates the ISR implicitly with AlterPartition requests. The controller will make sure than upon completion, the ELR only contains replicas in the final replica set. Additionally, in order to improve the durability of the reassignment
    a. The current behavior, when completing the reassignment, all the adding replicas should be in ISR. This behavior can result in 1 replica in ISR. Also, ELR may not help here because the removing ISR replicas can not stay in ELR when completed. So we propose to enforce that the reassignment can only be completed if the ISR size is larger or equal to min ISR.
    b. This min ISR requirement is also enforced when the reassignment is canceled.
5. Have a new admin API *DescribeTopicsRequest* for showing the topic details. We don't want to embed the ELR info in the *Metadata* API. The ELR is not some necessary details to be exposed to user clients.
    a. More public facing details will be discussed in the *DescribeTopicsRequest* section.
6. We also record the last-known ELR members.
    a. It basically means when an ELR member has an unclean shutdown, it will be removed from ELR and added to the *LastKnownELR*. The *LastKnownELR* will be cleaned when ISR reaches the min ISR.
    b. *LastKnownELR* is stored in the metadata log.
    c. *LastKnownELR* will be also useful in the Unclean Recovery section.
7. The last known leader will be tracked.
    a. This can be used if the Unclean recovery is not enabled. More details will be discussed in the Deliver Plan.
    b. The controller will record the last ISR member(the leader) when it is fenced.
    c. It will be cleaned when a new leader is elected.

Leader election

As the proposal changes a lot in our behaviors about the ISR, the leader election behavior will be described in detail in the Unclean Recovery section.

Detection of an unclean shutdown

The current log system will create a *CleanShutdownFile* after the log has flushed and right before shutdown. Then if the broker comes up again and finds this *CleanShutdownFile,* the broker can assume the log is complete after the reboot.

Based on *CleanShutdownFile,* we propose the following new behaviors.

1. During the shutdown, write the current broker epoch in the *CleanShutdownFile.*
2. During the start, the broker will try to read the broker epoch from the *CleanShutdownFile.* Then put this broker epoch in the broker registration request.
3. The controller will verify the broker epoch in the request with its registration record. If it is the same, it is a clean shutdown.
4. if the broker shuts down before it receives the broker epoch, it will write -1.

Note, the *CleanShutdownFile* is removed after the log manager is initialized. It will be created and written when the log manager is shutting down.

Unclean recovery

As the new proposal allows the ISR to be empty, the leader election strategy has to be reviewed.

- *unclean.leader.election.enable*=true, the controller will randomly elect a leader if the last ISR member gets fenced.
- *unclean.leader.election.enable*=false, the controller will only elect the last ISR member when it gets unfenced again.

Randomly electing a leader is definitely worth improving. As a result, we decide to replace the random election with the Unclean Recovery.

The Unclean Recovery uses a deterministic way to elect the leader persisted the most data. On a high level, once the unclean recovery is triggered, the controller will use a new API *GetReplicaLogInfo* to query the log end offset and the leader epoch from each replica. **The one with the highest leader epoch plus the longest log end offset will be the new leader**. To help explain when and how the Unclean Recovery is performed, let's first introduce some config changes.

The new *unclean.recovery.strategy* has the following 3 options.

Aggressive. It represents the intent of recovering the availability as fast as possible.
Balanced. Auto recovery on potential data loss case, wait as needed for a better result.
None. Stop the partition on potential data loss.

With the new config, the leader election decision will be made in the following order when the ISR and ELR meet the requirements:

1. If there are other ISR members, choose an ISR member.
2. If there are unfenced ELR members, choose an ELR member.
3. If there are fenced ELR members
    a. If the *unclean.recovery.strategy=Aggressive*, then an unclean recovery will happen.
    b. Otherwise, we will wait for the fenced ELR members to be unfenced.
4. If there are no ELR members.
    a. If the *unclean.recovery.strategy=Aggressive*, the controller will do the unclean recovery.
    b. If the *unclean.recovery.strategy=Balanced*, the controller will do the unclean recovery when all the *LastKnownELR* are unfenced. See the following section for the explanations.
    c. Otherwise, *unclean.recovery.strategy=None*, the controller will not attempt to elect a leader. Waiting for the user operations.

The controller will initiate the Unclean Recovery once a partition meets the above conditions. As mentioned above, the controller collects the log info from the replicas. Apart from the Log end offset and the partition leader epoch in the log, the replica also returns the following for verification:

- Replica's broker epoch. This can avoid electing a broker that has rebooted after it made the response.
- The current partition leader epoch in the replica's metadata cache. This is to fence stale *GetReplicaLogInfo* responses.

The controller can start an election when:

- In *Balance* mode, all the *LastKnownELR* members have replied, plus the replicas replied within the timeout. Due to this requirement, the controller will only start the recovery if the *LastKnownELR* members are all unfenced.
- In *Aggressive* mode, any replicas replied within a fixed amount of time OR the first response received after the timeout.

The behaviors in the failover:

- Broker failover.
    - If the replica fails before it receives the *GetReplicaLogInfo* request, it can just send the log info along with its current broker epoch.
    - If the replica fails after it responds to the *GetReplicaLogInfo* request
        - If the controller receives the new broker registration, the controller can reject the response because the broker epoch in the request mismatches with the broker registration.

        - Otherwise, the replica may become the leader but will be fenced later when it registers.
- Controller failover.
    - The controller does not store anything in the metadata log, every controller failover will result in a new unclean recovery.

Other

1. The kafka-leader-election.sh tool will be upgraded to allow manual leader election.
    a. It can directly select a leader.
    b. It can trigger an unclean recovery for the replica with the longest log in either *Aggressive* or *Balance* mode.
2. Configs to update. Please refer to the *Config Changes* section
3. For compatibility, the original *unclean.leader.election.enable* options *True/False* will be mapped to *unclean.recovery.strategy options.*
    a. *unclean.leader.election.enable.false -> unclean.recovery.strategy.Balanced*
    b. *unclean.leader.election.enable.true -> unclean.recovery.strategy.Aggressive*

Public Interfaces

We will deliver the KIP in phases, so the API changes are also marked coming with either ELR or Unclean Recovery.

PartitionChangeRecord (coming with ELR)

```
{
  "apiKey": 5,
  "type": "metadata",
  "name": "PartitionChangeRecord",
  "validVersions": "0-1",
  "flexibleVersions": "0+",
  "fields": [
...
// New fields begin.
    { "name": "EligibleLeaderReplicas", "type": "[]int32", "default": "null", "entityType": "brokerId",
      "versions": "1+", "nullableVersions": "1+", "taggedVersions": "1+", "tag": 6,
      "about": "null if the ELR didn't change; the new eligible leader replicas otherwise." }
    { "name": "LastKnownELR", "type": "[]int32", "default": "null", "entityType": "brokerId",
      "versions": "1+", "nullableVersions": "1+", "taggedVersions": "1+", "tag": 7,
      "about": "null if the LastKnownELR didn't change; the last known eligible leader replicas otherwise." }
    { "name": "LastKnownLeader", "type": "int32", "default": "null", "entityType": "brokerId",
      "versions": "1+", "nullableVersions": "1+", "taggedVersions": "1+", "tag": 8,
      "about": "-1 means no last known leader needs to be tracked." }
// New fields end.
  ]
}
```

PartitionRecord (coming with ELR)

```
{
  "apiKey": 3,
  "type": "metadata",
  "name": "PartitionRecord",
  "validVersions": "0-1",
  "flexibleVersions": "0+",
  "fields": [
...
// New fields begin.
    { "name": "EligibleLeaderReplicas", "type": "[]int32", "default": "null", "entityType": "brokerId",
      "versions": "1+", "nullableVersions": "1+", "taggedVersions": "1+", "tag": 1,
      "about": "The eligible leader replicas of this partition." }
    { "name": "LastKnownELR", "type": "[]int32", "default": "null", "entityType": "brokerId",
      "versions": "1+", "nullableVersions": "1+", "taggedVersions": "1+", "tag": 2,
      "about": "The last known eligible leader replicas of this partition." }
    { "name": "LastKnownLeader", "type": "int32", "default": "null", "entityType": "brokerId",
      "versions": "1+", "nullableVersions": "1+", "taggedVersions": "1+", "tag": 8,
      "about": "-1 means no last known leader needs to be tracked." }
// New fields end.
  ]
}
```

BrokerRegistration API (coming with ELR)

```
{
  "apiKey":62,
  "type": "request",
  "listeners": ["controller"],
  "name": "BrokerRegistrationRequest",
  "validVersions": "0-2",
  "flexibleVersions": "0+",
  "fields": [
...
// New fields begin.
    { "name": "PreviousBrokerEpoch", "type": "int64", "versions": "2+", "default": "-1",
      "about": "The epoch before a clean shutdown." }
// New fields end.
  ]
}
```

DescribeTopicPartitionsRequest (Coming with ELR)

Should be issued by admin clients. More admin client related details please refer to the Admin API/Client changes

ACL: Describe Topic

The caller can list the topics interested or keep the field empty if requests all of the topics.

Pagination.

This is a new behavior introduced. The caller can specify the maximum number of partitions to be included in the response.

If there are more partitions than the limit, these partitions and their topics will not be sent back. In this case, the Cursor field will be populated. The caller can include this cursor in the next request.

Note,

- There is also a server-side config to control the maximum number of partitions to return. *max.request.partition.size.limit*
- There is no consistency guarantee between requests.
- It is an admin client facing API, so there is no topic id supported.

```
{
  "apiKey": 74,
  "type": "request",
  "listeners": ["broker"],
  "name": "DescribeTopicPartitionsRequest",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "Topics", "type": "[]TopicRequest", "versions": "0+",
      "about": "The topics to fetch details for.",
      "fields": [
        { "name": "Name", "type": "string", "versions": "0+",
          "about": "The topic name", "versions": "0+", "entityType": "topicName"}
      ]
    },
    { "name": "ResponsePartitionLimit", "type": "int32", "versions": "0+", "default": "2000",
      "about": "The maximum number of partitions included in the response." },
    { "name": "Cursor", "type": "Cursor", "versions": "0+", "nullableVersions": "0+", "default": "null",
      "about": "The first topic and partition index to fetch details for.", "fields": [
      { "name": "TopicName", "type": "string", "versions": "0+",
        "about": "The name for the first topic to process", "versions": "0+", "entityType": "topicName"},
      { "name": "PartitionIndex", "type": "int32", "versions": "0+", "about": "The partition index to start
with"}
    ]}
  ]
}
```

DescribeTopicsResponse

```
{
  "apiKey": 74,
  "type": "response",
  "name": "DescribeTopicPartitionsResponse",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "0+", "ignorable": true,
      "about": "The duration in milliseconds for which the request was throttled due to a quota violation, or
zero if the request did not violate any quota." },
    { "name": "Topics", "type": "[]DescribeTopicPartitionsResponseTopic", "versions": "0+",
      "about": "Each topic in the response.", "fields": [
      { "name": "ErrorCode", "type": "int16", "versions": "0+",
        "about": "The topic error, or 0 if there was no error." },
      { "name": "Name", "type": "string", "versions": "0+", "mapKey": true, "entityType": "topicName",
"nullableVersions": "0+",
        "about": "The topic name." },
      { "name": "TopicId", "type": "uuid", "versions": "0+", "ignorable": true, "about": "The topic id." },
      { "name": "IsInternal", "type": "bool", "versions": "0+", "default": "false", "ignorable": true,
        "about": "True if the topic is internal." },
      { "name": "Partitions", "type": "[]DescribeTopicPartitionsResponsePartition", "versions": "0+",
        "about": "Each partition in the topic.", "fields": [
        { "name": "ErrorCode", "type": "int16", "versions": "0+",
          "about": "The partition error, or 0 if there was no error." },
        { "name": "PartitionIndex", "type": "int32", "versions": "0+",
          "about": "The partition index." },
        { "name": "LeaderId", "type": "int32", "versions": "0+", "entityType": "brokerId",
          "about": "The ID of the leader broker." },
        { "name": "LeaderEpoch", "type": "int32", "versions": "0+", "default": "-1", "ignorable": true,
          "about": "The leader epoch of this partition." },
        { "name": "ReplicaNodes", "type": "[]int32", "versions": "0+", "entityType": "brokerId",
          "about": "The set of all nodes that host this partition." },
        { "name": "IsrNodes", "type": "[]int32", "versions": "0+", "entityType": "brokerId",
          "about": "The set of nodes that are in sync with the leader for this partition." },
        { "name": "EligibleLeaderReplicas", "type": "[]int32", "default": "null", "entityType": "brokerId",
          "versions": "0+", "nullableVersions": "0+",
          "about": "The new eligible leader replicas otherwise." },
        { "name": "LastKnownELR", "type": "[]int32", "default": "null", "entityType": "brokerId",
          "versions": "0+", "nullableVersions": "0+",
          "about": "The last known ELR." },
        { "name": "OfflineReplicas", "type": "[]int32", "versions": "0+", "ignorable": true, "entityType":
"brokerId",
          "about": "The set of offline replicas of this partition." }]},
      { "name": "TopicAuthorizedOperations", "type": "int32", "versions": "0+", "default": "-2147483648",
        "about": "32-bit bitfield to represent authorized operations for this topic." }]
    },
    { "name": "NextTopicPartition", "type": "Cursor", "versions": "0+", "nullableVersions": "0+", "default":
"null",
      "about": "The next topic and partition index to fetch details for.", "fields": [
      { "name": "TopicName", "type": "string", "versions": "0+",
        "about": "The name for the first topic to process", "versions": "0+", "entityType": "topicName"},
      { "name": "PartitionIndex", "type": "int32", "versions": "0+", "about": "The partition index to start
with"}
    ]}
  ]
}
```

CleanShutdownFile (Coming with ELR)

It will be a JSON file.

```
{
  "version": 0
  "BrokerEpoch":"xxx"
}
```

ElectLeadersRequest (Coming with Unclean Recovery)

ACL: CLUSTER_ACTION

Limit: 1000 partitions per request. If more than 1000 partitions are included, only the first 1000 will be served. Others will be returned with *REQUEST_LIMIT_REACHED*.

```
{
  "apiKey": 43,
  "type": "request",
  "listeners": ["zkBroker", "broker", "controller"],
  "name": "ElectLeadersRequest",
  "validVersions": "0-3",
  "flexibleVersions": "2+",
  "fields": [
  ...
  { "name": "TopicPartitions", "type": "[]TopicPartitions", "versions": "0+", "nullableVersions": "0+",
    "about": "The topic partitions to elect leaders.",
    "fields": [
    ...

// New fields begin. The same level with the Partitions
     { "name": "DesiredLeaders", "type": "[]int32", "versions": "3+", "nullableVersions": "3+",
       "about": "The desired leaders. The entry should match with the entry in Partitions by the index." },
     },
// New fields end.

  ] },
  { "name": "TimeoutMs", "type": "int32", "versions": "0+", "default": "60000",
    "about": "The time in ms to wait for the election to complete." }
  ]
}
```

GetReplicaLogInfo Request (Coming with Unclean Recovery)

ACL: CLUSTER_ACTION

Limit: 2000 partitions per request. If more than 1000 partitions are included, only the first 1000 will be served. Others will be returned with *REQUEST_LIMIT_REACHED*.

```
{
  "apiKey":70,
  "type": "request",
  "listeners": ["broker"],
  "name": "GetReplicaLogInfoRequest",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "BrokerId", "type": "int32", "versions": "0+", "entityType": "brokerId",
        "about": "The ID of the broker." },
    { "name": "TopicPartitions", "type": "[]TopicPartitions", "versions": "0+", "nullableVersions": "0+",
    "about": "The topic partitions to query the log info.",
    "fields": [
      { "name": "TopicId", "type": "uuid", "versions": "0+", "ignorable": true, "about": "The unique topic ID"},
      { "name": "Partitions", "type": "[]int32", "versions": "0+",
        "about": "The partitions of this topic whose leader should be elected." },
    ]}
] }
```

GetReplicaLogInfo Response

```
{
  "apiKey":70,
  "type": "response",
  "name": "GetReplicaLogInfoResponse",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "BrokerEpoch", "type": "int64", "versions": "0+", "about": "The epoch for the broker." }
    { "name": "TopicPartitionLogInfoList", "type": "[]TopicPartitionLogInfo", "versions": "0+",
    "about": "The list of the log info.",
    "fields": [
      { "name": "TopicId", "type": "uuid", "versions": "0+", "ignorable": true, "about": "The unique topic
ID."},
      { "name": "PartitionLogInfo", "type": "[]PartitionLogInfo", "versions": "0+", "about": "The log info of a
partition.",
        "fields": [
      { "name": "Partition", "type": "int32", "versions": "0+", "about": "The id for the partition." },
      { "name": "LastWrittenLeaderEpoch", "type": "int32", "versions": "0+", "about": "The last written leader
epoch in the log." },
      { "name": "CurrentLeaderEpoch", "type": "int32", "versions": "0+", "about": "The current leader epoch for
the partition from the broker point of view." },
      { "name": "LogEndOffset", "type": "int64", "versions": "0+", "about": "The log end offset for the
partition." },
      { "name": "ErrorCode", "type": "int16", "versions": "0+", "about": "The result error, or zero if there
was no error."},
      { "name": "ErrorMessage", "type": "string", "versions": "0+", "nullableVersions": "0+", "about": "The
result message, or null if there was no error."}      ]},
    ]}
] }
```

kafka-leader-election.sh (Coming with Unclean Recovery)

```
...
// Updated field starts.
--election-type <[PREFERRED, UNCLEAN, LONGEST_LOG_AGGRESSIVE, LONGEST_LOG_BALANCED,
DESIGNATION]:
                                          Type of election to attempt. Possible
  election type>                          values are
                                          "preferred" for preferred leader election
                                          or "unclean" for a random unclean leader election,
                                          or "longest_log_agressive"/"longest_log_balanced" to choose the
replica
                                           with the longest log
                                          or "designation" for electing the given replica("desiredLeader") to
be the leader.
                                          If preferred election is selection, the
                                          election is only performed if the
                                          current leader is not the preferred
                                          leader for the topic partition. If
                                          longest_log_agressive/longest_log_balanced/designation
                                          election is selected, the
                                          election is only performed if there
                                          are no leader for the topic
                                          partition. REQUIRED.
--path-to-json-file <String: Path to    The JSON file with the list  of
  JSON file>                              partition for which leader elections
                                          should be performed. This is an
                                          example format. The desiredLeader field
                                          is only required in DESIGNATION election.

                                        {"partitions":
                                               [{"topic": "foo", "partition": 1, "desiredLeader": 0},
                                                {"topic": "foobar", "partition": 2, "desiredLeader": 1}]
                                        }
                                        Not allowed if --all-topic-partitions
                                          or --topic flags are specified.
// Updated field ends.
```

Config changes

The new configs are introduced for ELR

1. *eligible.leader.replicas.enabled*. It controls whether the controller will record the ELR-related metadata and whether ISR can be empty. False is the default value. It will turn true in the future.
2. *max.request.partition.size.limit.* The maximum number of partitions to return in a API response.

The new configs are introduced for Unclean Recovery.

1. *unclean.recovery.strategy.* Described in the above section. *Balanced* is the default value.
2. *unclean.recovery.manager.enabled. True* for using the unclean recovery manager to perform an unclean recovery. *False* means the random election will be used in the unclean leader election. False is the default value.
3. *unclean.recovery.timeout.ms.* The time limits of waiting for the replicas' response during the Unclean Recovery. 5 min is the default value.

New Errors

REQUEST_LIMIT_REACHED

As we introduce the request limit for the new APIs, the items sent in the request but over the limit will be returned with *REQUEST_LIMIT_REACHED*. It is a retriable error.

Admin API/Client changes

The admin client will start to use the DescribeTopicsRequest to describe the topic.

1. The client will split a large request into proper pieces and send them one after another if the requested topics count reaches the limit.
2. The client will retry querying the topics if they received the response with Cursor field.
3. The output of the topic describe will be updated with the ELR related fields.
4. TopicPartitionInfo will be updated to include the ELR related fields.

Metrics

The following metrics will be added for ELR

- kafka.controller.global_under_min_isr_partition_count. Gauge. It tracks the number of partitions with ISR size smaller than min ISR.

The following metrics will be added for Unclean Recovery

- kafka.controller.unclean_recovery_partitions_count. Gauge. It tracks the partitions that are under unclean recovery. It will be unset/set to 0 when there is no unclean recovery happening. Note, if in Balance mode, the members in LastKnownELR are not all unfenced, it is also counted as a live recovery.
- kafka.controller.manual_leader_election_required_partition_count. Gauge. It counts the partition that is leaderless and waits for user operations to find the next leader.
- kafka.controller.unclean_recovery_finished_count. Counter. It counts how many unclean recovery has been done. It will be set to 0 once the controller restarts.

Public-Facing Changes

1. The High Watermark will only advance if all the messages below it have been replicated to at least *min.insync.replicas* ISR members.
2. The consumer is affected when consuming the ack=1 and ack=0 messages. When there is only 1 replica(min ISR=2), the HWM advance is blocked, so the incoming ack=0/1 messages are not visible to the consumers. Users can avoid the side effect by updating the *min.insync.replicas* t o 1 for their ack=0/1 topics.
3. Compared to the current model, the proposed design has availability trade-offs:
    a. If the network partitioning only affects the heartbeats between a follower and the controller, the controller will kick it out of ISR. If losing this replica makes the ISR under min ISR, the HWM advancement will be blocked unnecessarily because we require the ISR to have at least min ISR members. However, it is not a regression compared to the current system at this point. But later when the network partitioning finishes, the current leader will put the follower into the pending ISR(aka "maximum ISR") and continue moving forward while in the proposed world, the leader needs to wait for the controller to ack the ISR change.
    b. Electing a leader from ELR may mean choosing a degraded broker. Degraded means the broker can have a poor performance in replication due to common reasons like networking or disk IO, but it is alive. It can also be the reason why it fails out of ISR in the first place. This is a trade-off between availability and durability.
4. The unclean leader election will be replaced by the unclean recovery.
5. For fsync users, the ELR can be beneficial to have more choices when the last known leader is fenced. It is worth mentioning what to expect when ISR and ELR are both empty. We assume fsync users adopt the *unclean.leader.election.enable* as false.
    a. If the KIP has been fully implemented. During the unclean recovery, the controller will elect a leader when all the *LastKnownElr* members have replied.
    b. If only the ELR is implemented, the *LastKnownLeader* is preferred when ELR and ISR are both empty.

Ack=0/1 comparison

Comparing the current ISR model with the proposed design

	Current	Proposed
Produce	• Ack=0. No ack is required. • Ack=1. Ack after writing to the local log	The same.
Consumer	Clients can consume incoming messages if the ISR size is below min ISR.	Clients can't consume new messages if the ISR size is below min ISR.
Replication	HWM will move forward even if the ISR size is below min ISR.	HWM can not move forward if the ISR size is below min ISR.
Recover when all replicas have been fenced	1. An unclean leader election is required if the last member in ISR can't come back. 2. Only the last ISR member can be elected as the leader. 3. The system will take all the data loss that the last ISR member has.	1. An unclean recovery is required if no suitable replica can be elected. 2. The member of ELR can be elected as the leader. No guarantee on the ack=0/1 messages but it will be the possible minimal.

Compatibility, Deprecation, and Migration Plan

High Watermark advance requirement

- *min.insync.replicas* will no longer be effective to be larger than the replication factor. For existing configs, the *min.insync.replicas* will be min(*min.insync.replicas,* replication factor).
- Cluster admin should update the *min.insync.replicas* to 1 if they want to have the replication going when there is only the leader in the ISR.
- Note that, this new requirement is not guarded by any feature flags/Metadata version.

ELR

It will be guarded by a new metadata version and the *eligible.leader.replicas.enabled*. So it is not enabled during the rolling upgrade.

After the controller picked up the new MV and *eligible.leader.replicas.enabled* is true, when it loads the partition states, it will populate the ELR as empty if the PartitionChangeRecord uses an old version. In the next partition update, the controller will record the current ELR.

MV downgrade: Once the MV version is downgraded, all the ELR related fields will be removed on the next partition change. The controller will also ignore the ELR fields.

Software downgrade: After the MV version is downgraded, a metadata delta will be generated to capture the current status. Then the user can start the software downgrade.

Clean shutdown

It will be guarded by a new metadata version. Before that, the controller will treat all the registrations with unclean shutdowns.

Unclean recovery

Unclean Recovery is guarded by the feature flag *unclean.recovery.manager.enabled*.

- For the existing *unclean.leader.election.enable*
    a. If true, *unclean.recovery.strategy* will be set to Aggressive.
    b. If false, *unclean.recovery.strategy* will be set to *Balanced.*
- *unclean.leader.election.enable* will be marked as deprecated.

Delivery plan

The KIP is a large plan, it can be across multiple quarters. So we have to consider how to deliver the project in phases.

The ELR will be delivered in the first phase. When the Unclean Recover has not shipped or it is disabled:

The main difference is in the leader election and the unclean leader election.

- If there are other ISR members, elect one of them.
- If there are other unfenced ELR members, elect one of them.
- If there are fenced ELR members
    - *unclean.leader.election.enable* false, then elect the first ELR member to be unfenced.
    - *unclean.leader.election.enable* true, start an unclean leader election.

- If there are no ELR members
    - *unclean.leader.election.enable* true, start an unclean leader election.
    - *unclean.leader.election.enable* false, then elect the *LastKnownLeader* once it is unfenced.

The unclean leader election will be randomly choosing an unfenced replica as it is today.

Test Plan

The typical suite of unit/integration/system tests. The system test will verify the behaviors of different failing scenarios.

The design will be verified with TLA+.

Rejected Alternatives

"Soft" strick min ISR

The idea is pretty similar to the proposed plan, here are the invariants:

- ISR will normally not be able to drop below ISR.
    - The controller does not remove a replica from ISR if the size of ISR <= minISR.
    - The leader does not remove lagging replicas from ISR if the size of ISR <= minISR.
    - The only exception is that the controller will kick out a replica regardless the size of the ISR if the replica joins after an unclean shutdown.
- The High Watermark can only advance if ISR size >= minISR.

Pros

- No extra metadata is required.
- The leader will send fewer ISR updates. Generally, if removing a replica will results the ISR below min ISR, which means the HWM can't advance, the leader will avoid updating the ISR.
- In a special case of the follower is network partitioning only from the controller, the replication will not be interrupted because fencing the replica does not result in ISR below minISR.

Cons

- The ISR now can have fenced replicas and lagging replicas. This makes a gap in the current operations.
    - Admin API needs to be adjusted. DescribeTopic API has to be sent to the leader to query the "clean" ISR(without lagging and fenced replica).
    - The operation on the topics like demote brokers requires "clean" ISR info to make sure the operation is safe.
    - The observability metrics need to be updated to reflect the "clean" ISR.
    - It may be confusing to the public that the proposal changes a well-known ISR concept.

Allow ISR shrink to empty + log-inspection-based unclean leader election

The idea is to allow ISR to shrink to size 0 if all the replicas are unavailable. This can ensure the last broker will not be automatically elected as leader after it comes back so that no silent data loss can happen. Then if we really lose all the replicas, we enforce an unclean leader election.

The second part of the design is the unclean leader election. Instead of choosing randomly from the replicas, the controller can query all the available brokers for their current log offset. Then the controller can choose the one with the longest log.

Pros

- It can save more ack=1 messages.
- It does not change any current invariants for the clients.

Cons

- We can't definitely claim the choice does not have data loss.
- The leader election only makes sense if enough replicas are unfenced. This is very likely to affect the overall service availability.

Current ISR + Sync phase + stability + shadow ISR

This proposal allows the last replica to become the leader even if it has an unclean shutdown. After it becomes the leader, it goes through a sync phase with a set of eligible followers for any lost data.

An eligible follower is defined by

1. It has not had an unclean shutdown.

2. It is a member of the shadow ISR. Shadow ISR is a superset of the ISR which follows the strict min ISR rule that its size will never drop below min ISR.

Pros

- It can save more ack=1 messages.
- It does not change any current invariants for the clients.
- It can claim no data loss for ack=all messages if there is an eligible follower to sync with.

Cons

- It may affect the service availability if we have to wait for another shadow ISR member to get back online.
- Implementation can be complicated.
- HWM can move backward.

Rejected proposal for the unclean recovery

Enhance the ListOffsets API

It is an option to enhance the ListOffsets API instead of creating a new API. The intention of the new API is to distinguish the admin API and the client API. To be specific, It is necessary to include the broker epoch in the recovery, but this piece of information does not necessarily to be exposed to the clients.

The controller elects the leader when a given number of replicas have replied

In *Balance* mode, instead of waiting for the last-known ELR members, the controller elects when receives a given number of responses. For example, if the replication factor is 3 and min-ISR is 2. If we require at least 2 responses, then controller will choose between the first 2 replicas.

However, it does not give enough protection if we don't require responses from all the replicas. Consider the following case. The ISR starts with [0,1,2]. Broker 2 falls behind and is kicked out of ISR. Then broker 1, and broker 0 suffers unclean shutdowns and broker 1 had a real data loss. Later, broker 2 and broker 1 come online and controller will choose between 1 and 2. Either option will have data loss.

Actually in this model, broker 2 is not likely to have the complete log, so just forcing a fixed number of responses does not improve much durability.