KIP-584: Versioning scheme for features

Status

Current state: Accepted

Discussion thread: here

Voting thread: here

JIRA: ⚠ Unable to render Jira issues macro, execution error.

Contents

Background

There are basically two somewhat separate areas that we're talking about within the scope of this KIP.

Client discovery

The first area is discovery on the client side. **How do clients of Kafka become aware of broker capabilities?** For example, how does the client find out if the broker supports a new type of event delivery semantics, or a new type of broker storage functionality? Note that here the "client" here may be a producer, a consumer, a command-line tool, a graphical interface, a third-party application built on top of Kafka (for purposes such as administration /monitoring), or may be even the broker itself trying to communicate with other brokers via RPCs.

Today, the client relies on the `ApiVersions{Request|Response}` to determine at a per-request level which version of request should be used to send to the brokers. This is sufficient for client-side logic that only relies on the specific versions of request, however for other logic that requires client-side behavior to be different as well, such request-level `ApiVersions` check would not be sufficient. Detailed examples can be found below.

Feature gating

Development in Kafka (from a high level) is typically organized into features, each of which is tracked by a separate version number. Feature gating tries to answer the question, **How does the broker decide what features to enable?** For example, When is it safe for the brokers to start serving new Exactly-Once(EOS) semantics? Should the controller send some additional information when it sends out `LeaderAndIsrRequest`? Should the broker write its log data in the "new" message format or one of the "old" ones?

Upon first look, it might seem like we should always enable the latest features. However, feature gating becomes necessary in some cases:

- Imagine the rolling upgrade of brokers, where, a Kafka cluster is being upgraded to use a new broker binary containing new versions of certain features. Until the binary upgrade completes successfully on all brokers, some brokers may support the new versions of the feature while some may not. This transitional phase can continue often for hours. During this time, any feature that would prevent successful communication and cooperation between the old and the new broker software must stay disabled. For example, the controller may not want to send out a new type of request, since the brokers which have not yet been upgraded would not understand this request.
- Next, after a certain version of a certain feature is available on all brokers (i.e. post upgrade), today there is no safety net. An incompatible broker that's not supporting an active version of the feature could still become active in the deployment. This could have bad consequences. For example, such a broker may be activated due to accidental operational errors causing an unintended downgrade/rollback. But the cluster doesn't protect itself towards preventing this accident.

Motivation

The above areas translate into separate problems within the scope of this KIP, as described below.

Client discovery

In general, today in Kafka, clients do not have a way to discover the min/max feature versions supported by a broker, or by **all** brokers in the cluster.

The `ApiVersionsRequest` enables the client to ask a broker what APIs it supports. For each API that the broker enumerates in the response, it will also return a version range it supports. It is an obvious limitation that the response schema doesn't provide any details about features and their versions. Even if it does, such a response may be insufficient for cases, where, a client needs to commence usage of a specific feature version in the cluster, based on whether **all** brokers in the cluster support the feature at a specific version. Consider the following problem. Users could upgrade their client apps before or after upgrading their brokers. If it happens before the broker upgrade, any new features on the client apps that depend on certain broker-side support (from **all** brokers in the cluster) must stay disabled, until all the brokers have also been upgraded to the required version. The question then becomes, at what point in time can these new features be safely enabled on the client side? Today, this has to be decided manually by a human, and it is error-prone. Clients don't have a way to programmatically learn what version of a certain feature is guaranteed to be supported by all brokers (i.e. cluster-wide finalized feature versions), and take suitable decisions.

For example, consider the Kafka Streams use case. For KIP-447, Streams would need to switch their clients to use the new thread level producer model only after **all** brokers support the same. But, switching to the new model at a time when **some** brokers don't support the new model, can have consequences that break EOS semantics. Note that, if only some brokers support the new model, then Streams can continue to still use the old per-task producer model, without switching to the new model — this is not a problem.

Feature gating

There is a configuration key in Kafka called `inter.broker.protocol` (IBP). Currently, the IBP can only be set by changing the static configuration file supplied to the broker during startup. Although the IBP started out as a mechanism to control the version of RPCs that the brokers used for communication with each other, it has grown since then to gate a huge number of other features. For example, the IBP controls whether the broker will enable static consumer group membership, whether it will enable exactly once functionality, and whether it will allow incremental fetch requests.

There are certain problems with the IBP configuration. The IBP assumes a one-dimensional spectrum of versions. Also, managing the `inter.broker.protocol` is cumbersome and error-prone. Changes to this configuration key requires a costly broker restart. This is the source of the infamous "double roll." If a customer wants to upgrade to the latest Kafka software and use the latest features, they must perform a rolling upgrade (first roll). But then, after the upgrade is complete, in order to update the IBP, they will need to modify every configuration file in the cluster and restart every broker (second roll) – this is a costly operation. Even worse, if they leave the IBP configuration key unset, it will default to the latest version, which will probably cause confusing cluster instability during the first roll.

Goals

The goals of this KIP are the following. For the above problems of client discovery and feature gating, we would like to propose a flexible, operationally easy, safe, and maintainable solution:

1. *Client discovery*: Provide an API to programmatically access the feature metadata. Here, the "metadata" refers to the feature version **levels** (i.e. the cluster-wide finalized **maximum** and **minimum** versions of broker features). We would like to serve this metadata to clients in an eventually consistent and scalable way.

2. *Feature gating*: Provide an API to safely, durably and dynamically finalize the upgrades to cluster-wide feature version levels. The API (and it's related tooling) can be used by the cluster operator (i.e. typically a human) to finalize feature **maximum** version level upgrades/downgrades.
3. *Rolling upgrades using single restart:* Rolling broker upgrades involving IBP bumps should be eventually possible with just a single rolling restart (instead of 2).

Explanation

1. *Client Discovery*:
    - By scalable, we mean the solution should horizontally scale to the metadata read and write workload as the cluster grows with more features, brokers and clients. It is expected that metadata discovery is read heavy, while write traffic is very low (feature version levels are finalized typically during releases or other configuration pushes, which, can happen few times a day per cluster).
    - The metadata served to the client will contain an epoch value. These epoch values are integers and will increase monotonically whenever the metadata changes. At any given time, the latest epoch for the metadata returned to the client is valid, and the cluster functions in accordance with the same. **Note:** here, the metadata epoch # applies for the entire metadata contents, and is not particularly related to the individual feature version – these are 2 separate things. Please read this section for more information.
    - We want to be careful and specific about what we mean by consistent here. The metadata we serve to the client during discovery, will be eventually consistent. It turns out that scaling to strongly consistent metadata reads is not easy (in the current Kafka setup). And the cost of a solution that's eventually consistent, can be made minimal, in the following way. Due to eventual consistency, there can be cases where an older lower epoch of the metadata is briefly returned during discovery, after a more recent higher epoch was returned at a previous point in time. We expect clients to always employ the rule that the latest received higher epoch of metadata always trumps an older smaller epoch. Those clients that are external to Kafka should strongly consider discovering the latest metadata once during startup from the brokers, and if required refresh the metadata periodically (to get the latest metadata).
2. *Feature gating*:
    - It's only allowed to modify **max** feature version level, using the newly provided API. The **min** feature version level can not be modified with the new API (see note on deprecation in Non-goals section).
    - By safe, we mean: when processing a request to finalize a set of feature version levels, the system will dynamically verify that all brokers in the cluster support the intended version. If not, the request will be rejected. Also, when a broker starts up, it will verify that it is compatible with the configured feature versions. If it is not, it will either refuse to start up or eventually die as soon as it discovers a feature incompatibility.
    - By durable, we mean that the finalized features should be persisted durably and remembered across broker restarts. Generally the system should tolerate faults, as much as it does for storage of other critical metadata (such as topic configuration).
    - By dynamic, we mean that the finalized features can be mutated in a cheap/easy way without compromising the uptime of broker /controller processes, or the health of the cluster.

Non-goals

Within the scope of this KIP, we provide only certain support related to feature downgrades and deprecation. These are described below:

1. *Downgrade of feature version level:*
   A feature "downgrade" refers to dropping support across the entire cluster for a feature version level. This means reducing the finalized **maximum** feature version level X to a version level Y, where Y < X, after the feature was already finalized at a newer version level X. Firstly, we leave it to the cluster operator (i.e. human) to decide whether the above actions are backwards compatible. It is **not within the scope of this KIP** to provide help to the cluster operator to achieve this step. After the cluster operator is past this step, we do provide the following support:
    a. Just like with upgrades, a downgrade request to reduce feature version level is rejected by the system, unless, all brokers support the downgraded versions of the feature. In the example above, the system expects all brokers to support the downgraded feature version Y.
    b. We assume that, downgrades of finalized max feature version levels, are rare. For safety reasons, to restrict user **intent**, we request for the user to specify an explicit "allow downgrade" flag (in the API request) to safeguard against easy accidental attempts to downgrade version levels. Note that despite setting this flag, certain downgrades may be rejected by the system if it is impossible.
2. *Deprecation of feature version level:*
    a. A need can arise to deprecate the usage of a certain version of one or more broker feature. A feature "deprecation" refers to increasing the finalized **minimum** feature version level X to a version level Y, where Y > X. Deprecating a feature version is an incompatible change, which requires a major release of Kafka. This is very unlike max feature version level upgrades, which can happen dynamically, after broker bits are deployed to a cluster.
    b. Firstly, the cluster operator (i.e. human) should use external means to establish that it is safe to stop supporting a particular version of broker feature. For example, verify (if needed) that no clients are actively using the version, before deciding to stop supporting it. It is **not within the scope of this KIP** to provide help to the cluster operator to achieve this step. After the cluster operator is past this step, we do provide the support that during a specific release of Kafka the system would mutate the persisted cluster-wide finalized feature versions to the desired value signaling feature deprecation.

Proposed changes

Below is a **TL;DR** of the changes:

- Each broker will advertise the version range of it's supported features in it's own `BrokerIdZnode` during startup. The contents of the advertisement are specific to the broker binary, but the schema is common to all brokers.
- The controller already watches `BrokerIdZnode` updates, and will serve as a gateway (via a newly introduced API) to safely finalize version upgrades to features – controller is well positioned to serve this requirement since it has a global view of all `BrokerIdZnode` in the cluster, and can avoid most race conditions in the solution.
- The metadata about cluster-wide finalized version levels of features, is maintained in ZK by the controller. As a general rule, any active feature version value is an int64 that's always >= 1. The value can be increased to indicate a version upgrade, or decreased to indicate a downgrade (or, if the value is < 1, it is considered as an ask for feature version level deletion).

- The controller will be the one and only entity modifying the information about finalized feature version levels. We expect metadata write traffic to be very low. All brokers in the cluster setup watches on the ZK node containing the finalized features information, thereby they get notified whenever the contents of the ZK node are change via the controller.
- Finally, client discovery about finalized feature versions will be performed via `ApiKeys.API_VERSIONS` API – the response will be modified to contain the necessary information (served from broker cache). Using this decentralized approach, we scale for metadata reads, albeit with eventually consistent results.

Broker advertisements

Each broker's supported dictionary of {feature version range} will be defined in the broker code. For each supported feature, the supported version range is defined by a `min_version` (an int64 starting always from 1) and `max_version` (an int64 >=1 and >= `min_version`).

The controller needs a way to discover this metadata from each broker. To facilitate the same, during startup, each broker will advertise it's supported dictionary of {feature version range} in it's own ephemeral `BrokerIdZnode` (this is the existing ZK node at the path `'/brokers/ids/<id>'`), under a nested dictionary keyed 'features'. The controller already watches all `BrokerIdZnode` for updates, and thus can get it's ZK cache populated with the per-broker versioning information (using existing means).

The schema for the advertised information is similar to the one in `'/features'` ZK node (see this section). Here is an example of a `BrokerIdZnode` with the proposed additional metadata towards the bottom.

BrokerIdZnode schema changes

```
{
   "listener_security_protocol_map":{
      "INTERNAL":"PLAINTEXT",
      "REPLICATION":"PLAINTEXT",
      "EXTERNAL":"SASL_SSL"
   },
   "endpoints":[
      "INTERNAL://kafka-0.kafka.def.cluster.local:9071",
      "REPLICATION://kafka-0.kafka.abc.cluster.local:9072",
      "EXTERNAL://b3-kfc-mnope.us.clusterx:9092"
   ],
   "rack":"0",
   "jmx_port":7203,
   "host":"kafka-0.kafka.def.cluster.local",
   "timestamp":"1579041366882",
   "port":9071,
   // ----- START: PROPOSED ADDITIONAL/MODIFIED METADATA -----
   "version":5,  // existing key whose value has been bumped by 1
   "features": {  // new key
      "group_coordinator": {  // string -> name of the feature
          "min_version": 1,   // int16 -> represents the min supported version (>=1) of this feature
          "max_version": 3  // int16 -> represents the max supported version of this feature (>=1 and >=
min_version)
      },
      "transaction_coordinator": {
          "min_version": 1,
          "max_version": 4
      }
   }
   // ----- END: PROPOSED ADDITIONAL METADATA -----
}
```

Persistence of finalized feature version levels

The proposal is that cluster-wide finalized max/min feature version levels will be persisted in a specific common ZK node. The path to the ZK node is proposed as: `'/features'` . The node content type is JSON (string), and the size is expected to be typically small (in several KBs). Couple high level details:

- During regular operations, the data in the ZK node can be mutated only via a specific admin API served only by the controller.
- An eventually consistent copy of the node data shall be made readable via existing `ApiKeys.API_VERSIONS` API served by any broker in the cluster (see this section). The latest copy of the node data will be cached in the controller, and if needed this can be obtained by directing the `ApiKeys.API_VERSIONS` API to the controller.

The above proposed read/write paths are described later in this doc. Below is an example of the contents of the `'/features'` ZK node, with it's schema explained below.

Schema (JSON)

```
{
   "version": 0, // int32 -> Represents the version of the schema for the data stored in the ZK node
   "status": 1, // int32 -> Represents the status of the node
   "features": {
               "group_coordinator": {   // string -> name of the feature
                        "min_version_level": 0, // int16 -> Represents the cluster-wide finalized minimum
version level (>=1) of this feature
                    "max_version_level": 3 // int16 -> Represents the cluster-wide finalized maximum version
level (>=1 and >= min_version_level) of this feature
            },
        "consumer_offsets_topic_schema": {
                   "min_version_level": 0,
                    "max_version_level": 4
        }
}
```

The schema is a JSON dictionary with few different keys and values, as explained below:

- The value for the `version` key is an int32 that contains the version of the schema of the data stored in the ZK node.
- The value for the `status` key is an int32 that contains the status of the ZK node (see explanation below).
- The value for the `features` key is a dictionary that contains a mapping from feature names to their metadata (such as finalized version levels). It's a `map{string  map{string  <string | number>}}`

  <feature_name>
  |--> <metadata_key>
  |--> <metadata_value>

  - Top-level key <feature_name> is a non-empty string that's a feature name.
    - <metadata_key> refers to the second nested level key that's a non-empty string that refers to some feature metadata.
    - <metadata_value> is either a string or a number – it's the value for the <metadata_key>.
    - **Note:** this nested dictionary would contain the following keys:
      - 'min_version_level' whose value is an int16 representing the **minimum** finalized cluster-wide version level for the feature.
      - 'max_version_level' whose value is an int16 representing the **maximum** finalized cluster-wide version level for the feature.
      - the following rule always holds true: min_version_level >= 1 and max_version_level >= 1 and min_version_level <= max_version_level.

Feature ZK node status

The `'/features'` ZK node schema contains a field: `status` that takes two values as explained below.

- `Enabled (1)`: This status means the feature versioning system is enabled, and, the finalized features stored in the `'/features'` ZK node are active. This status is written by the controller to the ZK node only when the broker IBP config is greater than or equal to `migration_ibp_version` (see migration section).
- `Disabled (0)`: This status means the feature versioning system is disabled, and, the finalized features stored in the `'/features'` ZK node is not relevant. This status is written by the controller to the ZK node only when the broker IBP config is less than `migration_ibp_version` (see migration section).

The purpose behind the status field is that it helps differentiates between the following cases:

1. New cluster bootstrap:
   For a new Kafka cluster (i.e. it is deployed first time), we would like to start the cluster with all the possible supported features finalized immediately. The new cluster will almost never be started with an old IBP config that's less than `migration_ibp_version`. In such a case, the controller will start up and notice that the `'/features'` is absent in the new cluster. To handle the requirement, the controller will create a `'/features'` ZK node (with enabled status) containing the entire list of supported features as its finalized features.
2. Cluster upgrade:
   Imagine there is an existing Kafka cluster with IBP config less than `migration_ibp_version`, but the Broker binary has been upgraded to a state where it supports the feature versioning system. This means the user is upgrading from an earlier version of the Broker binary. In this case, we want to start with no finalized features and allow the user to enable them whenever they are ready i.e. in the future whenever the user sets IBP config to be greater than or equal to `migration_ibp_version`. The reason is that enabling all the possible features immediately after an upgrade could be harmful to the cluster. In such a case:
   - Before the Broker upgrade (i.e. IBP config set to less than `migration_ibp_version`), the controller will start up and check if the `'/features'` ZK node is absent. If true, then it will react by creating a `'/features'` ZK node with disabled status and empty features.

- After the Broker upgrade (i.e. IBP config set to greater than or equal to `migration_ibp_versio`), when the controller starts up it will check if the `'/features'` ZK node exists and whether it is disabled. In such a case, it won't upgrade all features immediately. Instead it will just switch the `'/features'` ZK node status to enabled status. This lets the user finalize the features later.

3. Cluster downgrade:
   Imagine that a Kafka cluster exists already and the IBP config is greater than or equal to `migration_ibp_version`. Then, the user decided to downgrade the cluster by setting IBP config to a value less than `migration_ibp_version`. This means the user is also disabling the feature versioning system. In this case, when the controller starts up with the lower IBP config, it will switch the `'/features'` ZK node status to disabled with empty features.

Changes to Kafka Controller

We introduce 1 new Admin API that's served only by the Kafka Controller, and identified by the new API key: `ApiKeys.UPDATE_FEATURES`. This API enables application of a set of cluster-wide feature updates to the ZK `'/features'` node:

- The API requires AclOperation.ALTER on ResourceType.CLUSTER.
- The API request contains a list of `FeatureUpdate` that need to be applied, as explained below (see Validations section for more details):
  - Each item specifies the finalized feature to be added or updated or deleted, along with the new **max** feature version level value.
  - Downgrade or deletion of feature version level, is **not** a regular operation/intent. This is only attempted in the controller if the item sets an `allowDowngrade` flag, to convey user **intent** to attempt max version level downgrade/deletion. Note that despite this `allowDowngrade` flag being set, certain downgrades may be rejected by the controller if it is deemed impossible.
- The API response contains the result corresponding to each `FeatureUpdate` in the request. Each result is described by an error code and an error message.
- The API is **not** transactional, meaning that the request can be carried out partially i.e. some of the `FeatureUpdate` in the request can succeed, while the others don't.
- Changes to cluster-wide finalized **minimum** feature version level, can not be carried out using this API. This can be only done as explained later under Feature version deprecation section.

To help explain things better, below are the request and response definitions for the new API to update features (also see section showing related pseudocode for the Admin API):

UpdateFeaturesRequest schema

```
{
  "apiKey": 52,
  "type": "request",
  "name": "UpdateFeaturesRequest",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "timeoutMs", "type": "int32", "versions": "0+", "default": "60000",
      "about": "How long to wait in milliseconds before timing out the request." },
    { "name": "FeatureUpdates", "type": "[]FeatureUpdateKey", "versions": "0+",
      "about": "The list of updates to finalized features.", "fields": [
      {"name": "Feature", "type": "string", "versions": "0+", "mapKey": true,
        "about": "The name of the finalized feature to be updated."},
      {"name": "MaxVersionLevel", "type": "int16", "versions": "0+",
        "about": "The new maximum version level for the finalized feature. A value >= 1 is valid. A value < 1,
is special, and can be used to request the deletion of the finalized feature."},
      {"name": "AllowDowngrade", "type": "bool", "versions": "0+",
        "about": "When set to true, the finalized feature version level is allowed to be downgraded/deleted.
The downgraded request will fail if the new maximum version level is a value that's not lower than the existing
maximum finalized version level."}
    ]}
  ]
}
```

UpdateFeaturesResponse schema

Possible top-level errors:

- If the request was processed by a broker that's not the controller, then NOT_CONTROLLER error code is returned.
- If we didn't have sufficient permission to perform the update, then CLUSTER_AUTHORIZATION_FAILED error code is returned.

Individual FeatureUpdate errors:

- If the `FeatureUpdate` is invalid (ex: the provided `maxVersionLevel` is incorrect), then the result contains the INVALID_REQUEST error code.

- If the `FeatureUpdate` can not be applied (ex: due to version incompatibilities), then FEATURE_UPDATE_FAILED (a new error code) is returned.

```
{
  "apiKey": 52,
  "type": "response",
  "name": "UpdateFeaturesResponse",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "0+",
      "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": "ErrorCode", "type": "int16", "versions": "0+",
      "about": "The top-level error code, or `0` if there was no top-level error." },
    { "name": "ErrorMessage", "type": "string", "versions": "0+", "nullableVersions": "0+",
      "about": "The top-level error message, or `null` if there was no top-level error." },
    { "name": "Results", "type": "[]UpdatableFeatureResult", "versions": "0+",
      "about": "Results for each feature update.", "fields": [
      {"name": "Feature", "type": "string", "versions": "0+", "mapKey": true,
        "about": "The name of the finalized feature."},
      { "name": "ErrorCode", "type": "int16", "versions": "0+",
        "about": "The feature update error code or `0` if the feature update succeeded." },
      { "name": "ErrorMessage", "type": "string", "versions": "0+", "nullableVersions": "0+",
        "about": "The feature update error, or `null` if the feature update succeeded." }
    ]}
  ]
}
```

Feature advertisements from each broker in the cluster are reactively picked up by the controller via ZK watches. Today these watches are already set up by the controller on every broker's ephemeral ZK node (at '`/brokers/ids/<id>`'). The contents of the broker nodes are cached in memory in the Controller.

Underneath hood, the implementation of the `ApiKeys.UPDATE_FEATURES` API does the following:

1. For correctness reasons, the controller shall allow processing of only one inflight `ApiKeys.UPDATE_FEATURES` API call at any given time.
2. Prior to mutating `'/features'` in ZK, the implementation verifies that all broker advertisements (as seen by it thus far) contained no incompatibilities with the provided `List<FeatureUpdate>` (see Validations section below).
    a. Note that feature version deletions (using a version level < 1) can be used in circumstances where a finalized feature needs to be garbage collected. This type mutates '/features' in ZK without any guarding checks on feature versions.
3. Once validations in #2 are successful, the `List<FeatureUpdate>` are applied to contents of `'/features'` ZK node. Upon successful write, ZK should automatically increment the version of the node by 1.
4. Finally, an `UpdateFeatureResponse` is returned to the user.

Validations

For some <feature_name>, imagine that:

- X is the existing finalized feature max version level specified in `'/features'` ZK node.
- Y is the **new** finalized feature max version level, specified as part of some `FeatureUpdate`, in the `ApiKeys.UPDATE_FEATURES` API request.

Then, the API implementation allows a change from X to Y in the ZK node, only if one of the following cases is satisfied for the same `<feature_name>`:

- *Upgrade case:* Y > X and Y falls in the `[min_version, max_version]` range advertised by each live broker in the cluster.

    [OR]

- *Downgrade case: Y* < X, and Y falls in the `[min_version, max_version]` range advertised by each live broker, and `allowDowngrade` flag is set in the request.

    [OR]

- *Deletion case:* Y < 1 is special and indicates deletion of a finalized feature from the `'/features'` ZK node. It is allowed only if `allowDowngrade` flag is also set in the request.

Note the following **important** points:

1. The usual regular usage of the API is a feature upgrade. This is by specifying a higher new **max** feature version level value (i.e. `Y > X`).
2. By default, the API **disallows** max feature version level downgrades (`Y < X`) and finalized feature deletions (`Y < 1`). Such changes are allowed to be attempted only if the `allowDowngrade` flag is additionally set in the request. Note that despite the `allowDowngrade` flag being set, certain downgrades may be rejected by the controller, if the action is deemed impossible.
3. If any broker does not contain a feature present in the FeatureUpdate, this is considered an incompatibility such a case will fail the API request.

4. Some/all of the above logic will also be used by the broker (not just the controller) for it's protections (see this section of this KIP).

Guarantees

The `ApiKeys.UPDATE_FEATURES` API response contains a top-level result as well as result per individual `FeatureUpdate`.

- If the top-level error code indicates **success**, then it is guaranteed that there were no top-level failures. However, some individual `FeatureUpdate` may have still failed, so it is necessary to check the individual results.
- If the top-level error code indicates **failure**, then it is guaranteed that the request has failed:
  - A server-side error means the `'/features'` ZK node is guaranteed to be left unmodified i.e. no `FeatureUpdate` was applied.
  - A network error, or if the client dies during the request, or if the server dies when processing the request, it means the user should assume that the application of `List<FeatureUpdate>` may have either succeeded or failed.
- For each individual `FeatureUpdate` result indicating a **success** in the error code, the following is guaranteed:
  - The corresponding `FeatureUpdate` provided in the request was valid, and the update was applied to the `'/features'` ZK node by the controller (along with a bump to the ZK node version).
  - Brokers in the cluster have **gradually** started receiving notifications (via ZK watches) on the changes to '/features' ZK node. They react by reading the latest contents of the node from ZK, and re-establishing the ZK watch. This mechanism also enables brokers to shutdown when incompatibilities are detected (see Broker protections section).
- For each individual `FeatureUpdate` result indicating a **failure** in the error code, it is guaranteed that the corresponding `FeatureUpdate` was not applied to the `'/features'` ZK node by the controller.

Feature version deprecation

Deprecating a feature version is an incompatible change, which requires a major release of Kafka (see Non-goals section). This requirement translates to increasing the cluster-wide finalized **minimum** version level of one or more features in the ZK `'/features'` node. It is important to note that the **minimum** version level can not be mutated via the Controller API. This is because, the minimum version level is usually increased only to indicate the intent to **stop support** for a certain feature version. We would usually deprecate features during broker releases, after prior announcements. Therefore, this is not a dynamic operation, and such a mutation is not supported through the `ApiKeys.UPDATE_FEATURES` controller API.

Instead, it is best if the version deprecation is activated through the controller. This is the proposed approach to achieve the same:
Feature advertisements from each broker in the cluster are reactively picked up by the controller via ZK watches. Today these watches are already set up by the controller on every broker's ephemeral ZK node (at `'/brokers/ids/<id>'`). The contents of the broker nodes are cached in memory in the Controller. Using these contents, the controller should be able to track the highest supported feature minimum version exported by any broker in the cluster. Then, whenever a Kafka release advances the supported feature minimum version (to deprecate feature version), the controller picks up this change and persists the highest supported minimum version to ZK as the finalized **minimum** version level for the feature. This change by the controller will deprecate the finalized feature versions, unto 1 below the highest supported minimum version.

It is **important** that an admin should make sure that no clients are using a deprecated feature version (e.g. using the client version metric) before deploying a release that deprecates it.

Client discovery of finalized feature versions

The latest list of cluster-wide finalized feature version levels will be served via the existing `ApiKeys.API_VERSIONS` API. This is used for client discovery, as well as for debuggability of the system. We will introduce a tagged optional field in the schema of the `ApiVersionsResponse` to accommodate the information. Couple implementation details:

- Whenever ZK notifies the broker about a change to `'/features'` node, the broker will read the latest list of finalized features from ZK and caches it in memory (also see Broker protections section in this doc).
- We will modify the implementation of the `ApiKeys.API_VERSIONS` API to populate the information about finalized features in the response. This will contain the cluster-wide finalized versions from the broker's internal ZK cache of `'/features'` node contents (as explained above), as well as the broker-level supported versions (an additional piece of information that's useful for debugging).
- **The finalized feature versions populated in the response will be eventually consistent, since it is refreshed in the brokers via ZK watches.**

Below is new `ApiVersionsResponse` schema showing the two new tagged optional fields for feature versions.

Note that the field `FinalizedFeaturesEpoch` contains the latest version of the `'/features'` node in ZK. On the client side, the epoch can be used to ignore older features metadata returned in `ApiVersionsResponse`, after newer features metadata was returned once (eventual consistency).

ApiVersionsResponse schema (updated)

```
{
  "apiKey": 18,
  "type": "response", "name": "ApiVersionsResponse",
  "validVersions": "0-3",
  "flexibleVersions": "3+",
  "fields": [
    { "name": "ErrorCode", "type": "int16", "versions": "0+",
      "about": "The top-level error code." },
    { "name": "ApiKeys", "type": "[]ApiVersionsResponseKey", "versions": "0+",
      "about": "The APIs supported by the broker.", "fields": [
      { "name": "ApiKey", "type": "int16", "versions": "0+", "mapKey": true,
        "about": "The API index." },
      { "name": "MinVersion", "type": "int16", "versions": "0+",
        "about": "The minimum supported version, inclusive." },
      { "name": "MaxVersion", "type": "int16", "versions": "0+",
        "about": "The maximum supported version, inclusive." }]
    },
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "1+", "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." },
    // ----- START: PROPOSED ADDITIONAL METADATA -----
    { "name":  "SupportedFeatures", "type": "[]FeatureKey",
      "versions":  "3+", "tag": 10000, "taggedVersions": "3+",
      "about": "Features supported by the broker.",
      "fields":  [
        { "name": "Name", "type": "string", "versions": "3+",
          "about": "The name of the feature." },
        { "name": "MinVersion", "type": "int16", "versions": "3+",
          "about": "The minimum supported version for the feature." },
        { "name": "MaxVersion", "type": "int16", "versions": "3+",
          "about": "The maximum supported version for the feature." }
      ]
    },
    {"name": "FinalizedFeaturesEpoch", "type": "int64", "versions": "3+",
      "tag": 10001, "taggedVersions": "3+", "default": "-1", "ignorable": true,
      "about": "The monotonically increasing epoch for the features information."},
    { "name":  "FinalizedFeatures", "type": "[]FinalizedFeatureKey",
      "versions":  "3+", "tag": 10002, "taggedVersions": "3+",
      "about": "List of cluster-wide finalized features.",
      "fields":  [
        {"name": "Name", "type": "string", "versions":  "3+",
          "about": "The name of the feature."},
        {"name":  "MaxVersionLevel", "type": "int16", "versions":  "3+",
          "about": "The cluster-wide finalized max version level for the feature."},
        {"name":  "MinVersionLevel", "type": "int16", "versions":  "3+",
          "about": "The cluster-wide finalized min version level for the feature."}
      ]
    }
    // ----- END: PROPOSED ADDITIONAL METADATA -----
  ]
}
```

Broker protections against race conditions

Certain validations will be introduced at few points in the broker code. The purpose is to avoid race conditions where incompatible brokers remain active in a cluster. The validations affirm that the supported feature versions in the broker are compatible with the expected cluster-wide feature versions. If any of these checks fail, this would trigger the broker shutdown sequence, and the process eventually exits with a non-zero exit code. The places where the validation is going to be introduced, are explained below:

1. Validation shall be introduced during broker startup. This involves synchronously reading the cluster-wide feature versions from the `'/features'` ZK node just after initializing the ZK client, and before creating the broker's own ephemeral node (roughly here). The feature versions available in the broker are checked against the contents of the `'/features'` ZK node to ensure there are no incompatibilities. If an incompatibility is detected, the broker will be made to shutdown immediately.
2. A watch is setup on `'/features'` ZK node. Then, the above validation will be reused in the code path that reads the contents of the `'/features'` ZK node whenever a watch fires. This affirms that the feature versions available in the broker always remain compatible with the cluster-wide feature versions read from ZK.

NOTE: The logic for the validations will be exactly the same as the one described under Validations section under *Controller API*.

Incompatible broker lifetime race condition

Description of a rare race condition:

- T1: Imagine at time T1 the following event E1 occurs: A broker B starts up, passes feature validation, and registers it's presence in ZK in its `BrokerIdZnode`, along with advertising it's supported features. Assume that this is broker B is just about to become incompatible in it's feature set (due to event E2 below), in comparison to cluster-wide finalized feature versions.
- T1: At the same time T1, the following event E2 occurs that's concurrent with E1: a feature version level upgrade is finalized in the controller which causes broker B to become incompatible in it's feature set.
- T2: At a future time T2 the following event E3 occurs: The incompatible broker B receives the a ZK notification about a change to `'/features'` node. The broker validates the new contents of `'/features'` node against it's supported features, finds an incompatibility and shuts down immediately.

Question: The question is, what if the controller finalizes features via E2 while processing of broker registration via E1 is still in-flight/queued on the controller? Would this cause a harm to the cluster? Basically, in between T1 and T2, the broker B containing incompatible features could linger in the cluster. This window is very small (milli seconds), and typically rare – it can only happen in a rare case where an incompatible broker comes up in the cluster around the time that a feature version upgrade is finalized.

Solution: We intend to handle the race condition by careful ordering of events in the controller. In the controller, the thread that handles the `ApiKeys.UPDATE_FEATURES` request (E2) will be the `ControllerEventThread`. This is also the same thread that updates the controller's cache of Broker info whenever a new broker joins the cluster (E1). In this setup, if an `ApiKeys.UPDATE_FEATURES` request (E2) is processed ahead of a notification from ZK about an incompatible broker joining the cluster (E1), then the controller can certainly detect the incompatibility when it processes E1 after E2 (since it knows the latest finalized features). The controller would handle the incompatible broker, by skipping the handling of new broker registration. Then, it is only a matter of time (milli seconds) before the new broker receives a ZK notification (E3) about a change to `'/features'` node, then it automatically shuts itself down due to the incompatibility.

Tooling support

We shall introduce a `FeatureCommand` CLI tool backed by a new **admin** command library. This CLI tool will be maintained in the Kafka repository, alongside the code for the Kafka Broker. The section on Guidelines on feature versions and workflows, is a recommended read ahead of using the CLI tool in practice.

The CLI tool will be used by the **cluster operator (i.e. a human)**, and will enable us to do the following:

1. Read cluster-wide finalized feature versions from a broker or a controller via it's `ApiKeys.API_VERSIONS` API.
2. Add/update/delete **specific** or upgrade/downgrade **all** cluster-wide finalized feature versions by exercising the newly introduced `ApiKeys.UPDATE_FEATURES` API on a controller.

The CLI tool is versioned. Whenever a Kafka release introduces a new feature version, or modifies an existing feature version, the CLI tool will also be updated with this information. Newer versions of the CLI tool will be released as part of the Kafka releases. The tool is handy for both basic usage (upgrade /downgrade **all** feature max version levels) as well as advanced usage (change **specific** feature version levels). Later below, we demonstrate both such usages of the CLI tool.

- Basic usage happens typically when after a Kafka release, the cluster operator wants to finalize **all** latest feature versions. The tool internally has knowledge about a map of features to their respective max versions supported by the Broker. Using this information, the tool provides a facility to upgrade **all** feature max version levels to latest values known to the tool.
- Downgrade of **all** feature max version levels, **after** they are finalized using the tool, is a rare occurrence. To facilitate emergency downgrade of **all** feature versions (ex: just before emergency roll back to a previous Kafka release), the tool provides a `downgrade-all` facility. To achieve this, the user needs to run the version of the tool packaged with the Kafka release that he/she needs to downgrade to. This is because the tool's knowl edge of features and their version values, is limited to the version of the CLI tool itself (i.e. the information is packaged into the CLI tool when it is released).

We shall introduce 2 new APIs in the Admin interface, which enables us to read the feature versions and finalize feature version upgrades/downgrades. Below is Java-ish pseudocode for the same.

Admin API changes

```
// ---- START: Proposed Admin API definitions ----
/**
 * Describes supported and finalized features. You may anticipate
 * certain exceptions when calling get() on the future obtained from the returned
 * DescribeFeaturesResult.
 *
 * @param options   options for the describeFeatures API,
 *
 * @return          a result object containing:
 *                    1. List of cluster-wide finalized feature versions.
 *                    2. List of supported feature versions specific to the broker.
 */
DescribeFeaturesResult describeFeatures(DescribeFeaturesOptions options);

/**
 * Update the feature versions supported cluster-wide. You may
```

```
 * anticipate certain exceptions when calling get() on the futures
 * obtained from the returned UpdateFeaturesResult.
 *
 * @param updates   map of feature updates, keyed by the
 *                  name of the feature
 * @param options   options for the updateFeatures API
 *
 * @return          the results of the FeatureUpdate provided in the request
 */
UpdateFeaturesResult updateFeatures(Map<String, FeatureUpdate> featureUpdates, UpdateFeaturesOptions options);

/**
 * Options for {@link AdminClient#describeFeatures(DescribeFeaturesOptions)}.
 *
 * The API of this class is evolving.
 */
@InterfaceStability.Evolving
public class DescribeFeaturesOptions extends AbstractOptions<DescribeFeaturesOptions> {
  // Currently empty, but can be populated in the future as needed.
}

/**
 * Options for {@link AdminClient#describeFeatures(UpdateFeaturesOptions)}.
 *
 * The API of this class is evolving.
 */
@InterfaceStability.Evolving
public class UpdateFeaturesOptions extends AbstractOptions<UpdateFeaturesOptions> {
  // Currently empty, but can be populated in the future as needed.
}

// ---- END: Proposed Admin API definitions ----

// Represents a range of version levels supported by every broker in a cluster for some feature.
class FinalizedVersionRange {
    // The cluster-wide finalized value of the feature min version level (value >= 1).
    short minVersionLevel();

    // The cluster-wide finalized value of the feature max version level (value >=1 and value >=
minVersionLevel).
    short maxVersionLevel();
}

// Represents a range of versions that a particular broker supports for some feature.
class SupportedVersionRange {
        // The minimum version (value >= 1) of the supported feature.
        short minVersion();

        // The maximum version (value >=1 and value >= minVersion) of the supported feature.
        short maxVersion();
}

/**
 * Represents an update to a Feature, which can be sent to the controller
 * for processing.
 */
class FeatureUpdate {
    /**
     * The cluster-wide finalized NEW value of the feature max version level.
     * - When >= 1, it's the new value to-be-updated for the finalized feature.
         * - When < 1, it indicates the deletion of a finalized feature.
     */
    short maxVersionLevel();

    /**
     * Return true only if downgrade/deletion of a feature should be allowed.
     * Note that despite this allowDowngrade flag being set, certain downgrades
     * may be rejected by the controller if it is deemed unsafe to downgrade
     * the max version level of some specific feature.
     */
    bool allowDowngrade();
```

```
}

/**
 * Encapsulates details about finalized as well as supported features. This is particularly useful
 * to hold the result returned by the `Admin#describeFeatures(DescribeFeaturesOptions)` API.
 */
class FeatureMetadata {
    /**
     * Returns a map of finalized feature versions. Each entry in the map contains a key being a
     * feature name and the value being a range of version levels supported by every broker in the
     * cluster.
     */
        Map<String, FinalizedVersionRange> finalizedFeatures();

    /**
     * The monotonically increasing epoch for the finalized features.
     * If the returned value is empty, it means the finalized features are absent/unavailable.
     */
    Optional<Long> finalizedFeaturesEpoch();

        /**
     * Returns a map of supported feature versions. Each entry in the map contains a key being a
     * feature name and the value being a range of versions supported by a particular broker in the
     * cluster.
     */
    Map<String, SupportedVersionRange> supportedFeatures();
}

class DescribeFeaturesResult {
    /**
     * The data returned in the future contains the latest entire set of
     * finalized cluster-wide features, as well as the entire set of
     * features supported by the broker serving this read request.
     */
    KafkaFuture<FeatureMetadata> featureMetadata();
}

class UpdateFeaturesResult {
    /**
     * Returns a future which succeeds only if all the FeatureUpdate in the request succeed.
     */
    KafkaFuture<Void> all();

    /**
     * Returns a map with key being feature name and value being
     * the future which can be used to check the status of the FeatureUpdate
     * in the request.
     *
     * Possible error codes:
     * - NONE: The FeatureUpdate succeeded.
     * - NOT_CONTROLLER: The FeatureUpdate failed since the request was processed by a broker that's not the
controller.
     * - CLUSTER_AUTHORIZATION_FAILED: The FeatureUpdate failed since there wasn't sufficient permission to
perform the update.
     * - INVALID_REQUEST: The FeatureUpdate failed because it is invalid.
     * - FEATURE_UPDATE_FAILED: The FeatureUpdate failed because it can not be applied (ex: due to version
incompatibilities)
     */
    Map<String, KafkaFuture<Void>> values();
}
```

Basic CLI tool usage

Following are examples of regular usage of the CLI tool, which involves the following activities:

1. Read cluster-wide finalized feature versions from a broker or a controller via it's `ApiKeys.API_VERSIONS` API.
2. Upgrade the max version levels of **all** features, to their latest values, as known to the CLI tool internally. This becomes useful after completing the deployment of a new Kafka Broker release onto an existing cluster. This removes the burden to individually finalize feature upgrades.

3. Downgrade the max version levels of **all** features, to the values known to the CLI tool internally. This becomes useful during an emergency cluster downgrade, after finalizing feature levels from a previous upgrade.

```
=== DESCRIBE FEATURES ===

# Get cluster-wide finalized features, and features supported by a specific broker.
# - Use `--bootstrap-server` to provide a broker host:port to which queries should be issued.

$> kafka-features.sh
     --describe \
     --bootstrap-server kafka-broker0.prn1:9071 \

Feature: consumer_offsets_topic_schema  SupportedMinVersion: 1  SupportedMaxVersion: 1
FinalizedMinVersionLevel: -     FinalizedMaxVersionLevel: -     Epoch: 1
Feature: group_coordinator  SupportedMinVersion: 1  SupportedMaxVersion: 2  FinalizedMinVersionLevel: 1
FinalizedMaxVersionLevel: 1     Epoch: 1
Feature: transaction_coordinator  SupportedMinVersion: 1  SupportedMaxVersion: 5  FinalizedMinVersionLevel:
1     FinalizedMaxVersionLevel: 4     Epoch: 1

=== UPGRADE TO ALL LATEST FEATURES ===

# Upgrade to the max version levels of all features, as internally known to the CLI tool.
#
# - This command removes the burden to individually finalize feature upgrades.
#   This becomes handy to a cluster operator intending to finalize a cluster with all the latest
#   available feature version levels. This usually happens after completing the deployment
#   of a newer Kafka Broker release onto an existing cluster.
# - Use `--bootstrap-server` to provide a broker host:port to which queries should be issued.
# - Optionally, use the `--dry-run` flag to list the feature updates without applying them.

$> kafka-features.sh \
     --upgrade-all \
     --bootstrap-server kafka-broker0.prn1:9071 \
     [--dry-run]

    [Add] Feature: consumer_offsets_topic_schema       ExistingFinalizedMaxVersion: -  NewFinalizedMaxVersion:
1       Result: OK
[Upgrade] Feature: group_coordinator      ExistingFinalizedMaxVersion: 1  NewFinalizedMaxVersion: 2
Result: OK
[Upgrade] Feature: transaction_coordinator      ExistingFinalizedMaxVersion: 4  NewFinalizedMaxVersion: 5
Result: OK

=== EMERGENCY DOWNGRADE ALL FEATURES ===

# Downgrade to the max version levels of all features known to the CLI tool.
#
# - This command removes the burden to individually finalize feature version
#   downgrades. This becomes handy to a cluster operator intending to downgrade all
#   feature version levels, just prior to rolling back a Kafka Broker deployment
#   on a cluster, to a previous Broker release.
# - Optionally, use the `--dry-run` flag to list the feature updates without applying them.

$> kafka-features.sh \
     --downgrade-all \
     --bootstrap-server kafka-broker0.prn1:9071 \
     [--dry-run]

   [Delete] Feature: consumer_offsets_topic_schema       ExistingFinalizedMaxVersion: 1
NewFinalizedMaxVersion: -       Result: OK
[Downgrade] Feature: group_coordinator      ExistingFinalizedMaxVersion: 2  NewFinalizedMaxVersion: 1
Result: OK
[Downgrade] Feature: transaction_coordinator      ExistingFinalizedMaxVersion: 5  NewFinalizedMaxVersion:
4       Result: OK
```

Advanced CLI tool usage

Following are examples of advanced usage of the CLI tool. Going beyond regular usage, advanced usage involves adding/upgrading/downgrading/deleting **specific** cluster-wide finalized feature versions.

```
=== ADD OR UPGRADE OR DOWNGRADE OR DELETE FEATURES ===

# Add or update a list of cluster-wide finalized features.
# - Use `--bootstrap-server` to provide a broker host:port to which the queries should be issued.
# - Optionally, use `--upgrade` to provide a comma-separated list of features and new finalized max version to
add or upgrade.
# - Optionally, use `--downgrade` to provide a comma-separated list of features and new finalized max version
to downgrade to. This should be used only when required.
# - Optionally, use `--delete` to provide a comma-separated list of finalized features to be deleted.
# - Optionally, use the `--dry-run` flag to list the feature updates without applying them.

$> kafka-features.sh update \
     --bootstrap-server kafka-broker0.prn1:9071 \
     --upgrade group_coordinator:2,consumer_offsets_topic_schema:1 \
     --downgrade transaction_coordinator:3 \
     --delete replication_throttling \
     [--dry-run]

      [Add] Feature: consumer_offsets_topic_schema       ExistingFinalizedMaxVersion: -
NewFinalizedMaxVersion: 1       Result: OK
  [Upgrade] Feature: group_coordinator      ExistingFinalizedMaxVersion: 1  NewFinalizedMaxVersion: 2
Result: OK
[Downgrade] Feature: transaction_coordinator      ExistingFinalizedMaxVersion: 4  NewFinalizedMaxVersion:
3       Result: OK
   [Delete] Feature: replication_throttling       ExistingFinalizedMaxVersion: 2  NewFinalizedMaxVersion:
-       Result: OK
```

New or changed public interfaces

Summary of changes:

1. We introduce 1 new API in the broker RPC interface (see this section). This is the new `ApiKeys.UPDATE_FEATURES` (schema) that can only be served successfully by the controller.
2. We introduce few optional fields in the `ApiVersionsResponse` containing the cluster-wide finalized feature metadata, feature metadata epoch, and the broker's supported features (see this section).
3. We introduce 2 new APIs in the Admin interface to describe/addOrUpdate/delete features (see this section). Underneath covers, these exercise the APIs mentioned above.

Guidelines on feature versions and workflows

With the newly proposed feature versioning system in this KIP, it becomes quite important to understand when to use it, and when not to.

When to use versioned feature flags?

1. A "breaking change" typically happens whenever the underlying Broker logic has introduced a different concept (such as a message type, or a new field in a protocol) which only certain newer versions of the Broker code can understand/process. The older Broker versions could consider such a change to be alien i.e. the older Broker versions could exhibit undefined or terminal behavior if they notice effects of the breaking change. Similarly, Kafka clients could exhibit undesirable behavior if they commence usage of a feature version on the cluster, assuming all Brokers support it, while in reality some Brokers don't.
   As part of Kafka operations, there are occasions when it becomes necessary to **safely** activate certain breaking changes introduced by a newer Broker version. Such changes become good candidates to be released behind a feature flag . The cluster-wide max feature version level can be safely increased/finalized (using the provided mechanism in this KIP), after the cluster-wide deployment of the required Broker version is over. With the versioning system in place, whenever the effects of the underlying breaking change is activated by the cluster operator, the system protects the cluster and Kafka clients from bad effects due to breaking changes.
2. As a guideline, max feature version values should be increased if and only if the change is a breaking change. Also, min feature version value should be increased if and only if a feature version needs to be deprecated. Non-breaking changes or non-deprecating changes could continue to be made to the Broker code without modifying the feature version values.
3. As a guideline, whenever each feature's supported version range is modified, or if feature versions are permanently deprecated, these should be documented in the upgrade section of Kafka release notes.

What are the common workflows?

1. *Upgrades:* Most common workflow for the feature versioning system would involve finalizing a cluster upgrade to max feature version levels for **all** features, after completing the release of a new Kafka Broker binary on a cluster. This is explained under Basic CLI tool usage section.
2. *Downgrades:* Emergency downgrades of max version levels of features are rare, especially after they are finalized (because these are breaking changes, and finalizing these changes can have consequences). If the user has set the 'allowDowngrade' boolean field in an UpdateFeaturesRequest, then they must have had a good reason to attempt downgrading. Also, the CLI tool only allows a downgrade in conjunction with specific flags and sub-commands, as explained below:

a. In the CLI tool, if the user passes '--allow-downgrade' flag when updating a specific feature, then the tool will translate this ask to setting 'allowDowngrade' field in the request to the server.
b. Emergency downgrade of **all** features can be attempted using the CLI tool belonging to a specific Broker release. This is by using the 'downgrade-all' sub-command. For examples of these, please refer to the Tooling support section.
c. **Note:** Please note that some of the downgrades may still be blocked by Controller if it is deemed impossible to downgrade. The controller (on it's end) can maintain rules in the code, that, for safety reasons would outright reject certain downgrades from a specific max_version_level for a specific feature. Such rejections may happen depending on the feature being downgraded, and from what version level.

Use case: `group_coordinator` feature flag (for KIP-447)

As described in the motivation for this KIP, the Kafka Streams use case is going to be immediately benefitted by programmatically learning the cluster-wide version of EOS feature on the brokers (KIP-447). We shall use the `group_coordinator` feature flag to achieve the same, as explained below. This feature flag will cover both schema and protocol changes related to group coordinator (KIP-447 falls under a change to the group coordinator).

Imagine we shipped the feature versioning scheme before the shipment of KIP-447. There would then be at least two versions of the broker binary:

- One of these (the old/existing broker bits) would only advertise `group_coordinator` feature with max version 1 (shortly referred to as `v1` below) . `v1` doesn't contain the latest EOS semantic described in KIP-447.
- The other (the new broker bits) would advertise `group_coordinator` feature `v1-v2` i.e. max version `v1` as well as `v2`. `v2` contains the EOS semantic described in KIP-447.

Kafka Streams client uses the consumer rebalance protocol to propagate the group metadata information. There would be one nominated leader among the group. The sequence of feature upgrade events in the happy path shall be the following (T_0, T_1, ..., T_n refers to increasing time):

- **T_0:** The broker cluster is running with old binary, and a stream application is communicating with it. The stream clients will be using `group_coordinator:v1` feature. The leader stream thread would use its admin client to periodically call `DescribeFeatures` on a random broker in the cluster (presumably every minute), to get information about the latest cluster-wide enabled features. Thus, it would learn about the `group_coordinator` feature flag, with the finalized version at `v1` .
- **T_1:** The cluster undergoes a rolling upgrade to the new broker binary. Once the upgrade is complete, every broker in the cluster is expected to support `v1-v2` versions for the `group_coordinator` feature flag.
- **T_2:** The controller still will has enabled only `group_coordinator:v1` in `'/features'` ZK node. So, the above rolling has no effect (yet) on the streams client side.
- **T_3:** Knowing that the cluster upgrade has completed, the cluster operator (i.e. a human) sends an `ApiKeys.UPDATE_FEATURES` request to the controller. This is to finalize the cluster-wide `group_coordinator` feature version level upgrade from `v1` to `v2`. When this request is successful, it means the following:
  The controller has checked all brokers in the cluster advertised support for `group_coordinator:v2`, and it has persisted the upgrade to ZK `'/features'` node.
  The brokers are gradually receiving ZK notifications about the update to the `'/features'` ZK node. When each broker refreshes the contents of the ZK node, it will become aware that `group_coordinator:v2` has been finalized cluster-wide.
- **$T_{3\text{-}4}$:** The streams leader thread, at a certain point in time will see the above change (via the periodic `DescribeFeatures` call). It will then trigger a rebalance of the streams group plumbing down `group_coordinator:v2` feature flag to all members. It is key to note that this event could happen concurrently with **T_3** that's why the event is labelled **$T_{3\text{-}4}$**.
- **T_5:** Every member of the streams group will be gradually upgrading towards `group_coordinator:v2` just as the leader instructed above. This is when the stream client will switch from the old per-task producer model to the new thread level producer model (for KIP-447).

There are some edge cases in this happy upgrade path we shall explain below.

Delay of feature version propagation

Due to eventual consistency in metadata propagation, it is possible that some brokers get notified about `group_coordinator:v2` later than others. At **$T_{3\text{-}4}$**, the streams leader thread first upgrades to `group_coordinator:v2` when it learns of a feature version level bump from the broker. Then, after **T_5** , there exists a case where the leader streams thread could query a stale broker, and learn that the feature support only contains `v1`. If this situation is not handled properly, we are in a danger of downgrade. This is because in KIP-447 we don't support downgrade of the processing model, unless user wipes out the old stream instances and restarts from scratch. Following is done to make sure the above scenario never affects the stream use case:

1. The stream clients shall not downgrade to `group_coordinator:v1` once they upgrade to `group_coordinator:v2`. The solution is to persist the `group_coordinator` flag as part of the consumers' metadata when the rebalance completes, so that on a second rebalance as long as some members are already on the newer version, the leader of the group **will not kick off another feature request** because it knows an upgrade was ongoing in last generation, based on the eventual consistency guarantee. It will just continue to inform all the members to upgrade to the newer version.
2. It is important to note that the user is only upgrading the cluster, from enabling `group_coordinator:v1` to enabling `group_coordinator:v1-v2`. Therefore, once a client sees that `v2` is enabled on any broker, it should be a clear signal that **all broker binaries within the cluster would support `group_coordinator:v2` semantic, it's just a matter of time before feature version update metadata propagates to every broker from controller.**

The above ensures that the upgrade logic is reliable/safe, as there is no case of a downgrade affecting the stream client (during the upgrade path).

Downgrade of `group_coordinator` feature

As such, we don't support downgrades in this versioning scheme/system (see Non-goals and Future work). So, there is a danger if the cluster operator tries to downgrade the `group_coordinator` feature flag from `v2` to `v1` using the `--allow-downgrade` flag. This is because the stream client side shall

not downgrade at all, and this would crash the EOS applications. A safe practice is to first downgrade the stream clients, before downgrading the `group_coordinator` feature flag.

Potential features in Kafka

There could be many features benefitted by the above system, following are a selected few high level examples:

1. `group_coordinator`: As described in the above section.
2. `transaction_coordinator`: This feature flag could cover changes to the message format for the transaction state internal topic schema, and protocol changes related to transaction coordinator. The advantages are similar to #2.
3. `consumer_offsets_topic_schema`: Currently the message format used in the consumer internal topics is tied to IBP. It's a candidate for a feature, because we will be able to dynamically finalize a new version of the topic schema, without having to roll the brokers twice.
4. `inter_broker_protocol`: For transitional purposes, the `inter.broker.protocol` itself can be made a coarse-grained feature flag. This can be a way to operationally migrate away avoiding the double roll during IBP bumps.

Deployment, IBP deprecation and avoidance of double rolls

There is a configuration key in Kafka called `inter.broker.protocol` (IBP). Currently, the IBP can only be set by changing the static configuration file supplied to the broker during startup. Here are the various phases of work that would be required to use this KIP to eventually avoid Broker double rolls in the cluster (whenever IBP values are advanced).

Phase #1

Post development of the feature versioning system described in this KIP, we hit phase #1 where the new versioning system is ready to be used:

- The feature versioning system will be released under a new IBP version: `migration_ibp_version`.
- For the initial roll out, we should only operate the versioning system after the second IBP roll this brings the IBP of the cluster to `migration_ibp_version` (that's when versioning system is fully deployed).
    - As a safeguard, each broker will validate that it's IBP version is at least at `migration_ibp_version` before applying broker validations for feature versions, and before advertising it's features in ZK.
    - As a safeguard, the controller will validate that it's IBP version is at least at `migration_ibp_version` before allowing for feature version changes to be finalized in a cluster.

Phase #2

This is the phase when both the new feature versioning system as well as the existing IBP setup (in the form of static configuration) are active in the Kafka broker. Feature flags may be optionally defined in the code as part of Kafka development. During deployment of newer Kafka releases that use feature flags, there will be a requirement to finalize such feature flags using the provided API/tooling (as required by this KIP). By the end of this phase, we still have not eliminated the requirement for double roll during broker upgrades.

Phase #3

This phase is about realizing the benefit of this KIP to avoid Broker double rolls during upgrades. For this, we would like to move away from using the existing IBP-based setup (in the form of a static configuration) in the broker code base. This requires several steps, as proposed below:

1. We need a way to map the usage of IBP in the code (in the form of a static configuration) to the usage of IBP in the new feature versioning system. To achieve this, we introduce one or more feature flags in the code. These will be used to release features which were otherwise released under the static IBP config. We illustrate the write-up below using one such feature flag called as `ibp-feature`. For example, we will use the `ibp-feature` flag in the code at places wherever newer IBP values (from static configuration) are otherwise needed to be used:
    a. The max version values for this flag will start from 1 and continue increasing for future IBP version advancements.
    b. The min version value for this flag will start from 1, and it is unlikely to be modified (since we rarely or almost never deprecate IBP versions).
2. By this point, IBP-based decision making in the broker code will be such that:
    a. If the `ibp-feature` flag is finalized and if static IBP config value is >= `migration_ibp_version` , then the value of the `ibp-feature` flag is preferred for decision making over the IBP value from static configuration.
    b. Otherwise if the `ibp-feature` flag is not finalized yet, we continue to use the latest IBP value based on static configuration for decision making.
3. We then release the `ibp-feature` flag as part of a subsequent Kafka release. The release would eventually get deployed to Kafka clusters, and, the `ibp-feature` flag is expected to be finalized in the cluster (via provided tooling).
4. Once #3 happens, all future Kafka code changes can continue to use the `ibp-feature` flag, thereby effectively stopping the use of IBP as a static configuration.

Phase #4

This is the phase when we no longer advance the IBP values in the old IBP-based setup (in the form of static broker configuration) and have completely switched to `ibp-feature` flag(s) in the Kafka code. The former will be kept around for legacy purposes only.

Future work

As part of future work, we could consider adding better support for downgrades & deprecation. We may consider providing a mechanism through which the cluster operator could learn whether a feature is being actively used by clients at a certain feature version. This information can be useful to decide whether the feature is safe to be deprecated. Note that this could involve collecting data from clients about which feature versions are actively being used, and the scheme for this needs to be thought through.

Rejected alternatives

1. We considered the idea of using the existing `AlterConfigsRequest` instead of introducing a new API to update features. Reasons to decide against using it:
    a. The AlterConfigsRequest format is not well equipped to elegantly express the notion of `Set<FeatureUpdate>` operations. The request format is more favorable towards expressing a list of key-value pairs, where each new value replaces the existing value for the specified key. Furthermore, the response format doesn't let us conveniently return the list of finalized feature versions to the caller.
    b. AlterConfigsRequest can be issued to any broker, which is not our requirement. For the future, anything that modifies ZK needs to go through the controller. This is part of the work that we are doing to build the bridge release for KIP-500.
2. As opposed to fine-granular features, we considered the idea of exposing the notion of IBP directly to the client for discovery. There are a few issues here:

    a. We are introducing a leaky abstraction here. Users/clients (excluding clients that are brokers) have to now operate with the notion of a "broker protocol version" – this is merely a detail that's internal to a cluster of brokers behind an API.
    b. This still doesn't solve the "double roll" problem related to IBP. We would need to still solve this problem.