KIP-778: KRaft to KRaft Upgrades

Status

Current state: *Adopted*

Discussion thread: https://lists.apache.org/thread.html/rf8947e0d6aad51023b378305acc285c69030988abc7bda9b9c429b8a%40%3Cdev.kafka.apache.org%3E

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

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

Motivation

In order to facilitate rolling upgrades of Kafka in KRaft mode, we need the ability to upgrade controllers and brokers while holding back the use of new RPCs and record formats until the whole cluster has been upgraded. We would also like to perform this upgrade without the need for the infamous “double roll”.

Additionally, we want the ability to perform downgrades in a controlled and predictable manner.

Background

IBP

Kafka clusters currently use the IBP (inter.broker.protocol.version) configuration to gate new features and RPC changes. Since this is a static configuration defined for each broker in its properties file, a broker must be restarted in order to update this value. This leads to the necessity of two rolling restarts for cluster upgrades:

- Restart each broker with latest binary

- Restart each broker with new IBP

Support for downgrading the IBP is also poorly defined. For versions prior to Kafka 3.0, downgrading the IBP is explicitly not supported (based on the documentation). There have been cases where a persisted format was gated by the IBP and downgrades are impossible. However, in practice, we see that for many versions it can often be done without any problems.

ApiVersions

Brokers and Controllers advertise their API capabilities using ApiVersions RPC. In some cases, the advertised versions of a node's RPCs are influenced by broker's configured record version or IBP (inter.broker.protocol.version). However, in most cases if an RPC version is defined in the binary of a particular node, it will be advertised through ApiVersions.

One exception is when a broker is advertising controller-forwarded APIs (KIP-590). In this case, a broker connects to the active controller to learn its ApiVersions. For forwarded APIs, the broker and controller's ApiVersions are intersected to determine maximal safe set of APIs to advertise.

ApiVersionResponse includes:

- ApiKeys and the min/max version supported by the broker
- Supported features and the min/max versions for the broker
- Finalized (cluster-wide) features and the min/max version for the cluster

Feature Flags (KIP-584)

Supported feature flags depend on the version of the code and represent the capabilities of the current binary. A broker defines a minimum and maximum supported version for each feature flag. Finalized feature flags are dynamic and set by an operator using the "kafka-features.sh" script. The operator defines a maximum finalized version for the given feature flag which is used to convey the "in-use" version of the feature within the cluster. To date, no features in Kafka are utilizing these feature flags.

KRaft Snapshots

When a KRaft client joins the quorum and begins fetching, the leader may determine that it needs to load a snapshot to catch up more quickly. This snapshot includes the entire metadata state at a given point in time and will effectively force a broker or controller to re-build it's internal data structures which derive from the metadata log. For example, on the broker, processing a snapshot will result in a new MetadataImage which is the backing data structure for MetadataCache.

Public Interfaces

Define new KIP-584 feature flag "***metadata.version***". This feature flag is only examined or initialized in KRaft mode. In ZK mode, this feature flag is ignored.

Replace "AllowDowngrade" with "DowngradeType" in UpdateFeaturesRequest. Also add new "DryRun" field to correspond with the existing `--dry-run` flag.

```
{
  "apiKey": 57,
  "type": "request",
  "listeners": ["zkBroker", "broker", "controller"], <-- New listener "controller" for KRaft
  "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."},
      ------------------ Remove Field -------------
      {"name": "AllowDowngrade", "type": "bool", "versions": "0+",
        "about": "When set to true, the finalized feature version level is allowed to be downgraded/deleted.
The downgrade request will fail if the new maximum version level is a value that's not lower than the existing
maximum finalized version level."}
      ------------------ New Field ----------------
      {"name": "DowngradeType", "type": "int8", "versions": "0+", "default": 0,
        "about": "The type of downgrade to perform. Three types are supported: 0 is NONE (no downgrade will be
performed), 1 is SAFE, and 2 is UNSAFE. The safety of a downgrade is determined by the controller and is
specific to each feature flag."}
      ------------------ End New Field ------------
    ]},
    ------------------ New Field ------------------
    {"name": "DryRun", "type": "bool", "versions": "1+", "default": false},
    ------------------ End New Field --------------
  ]
}
```

One new code for UpdateFeaturesResponse results ErrorCode

- *UNSAFE_FEATURE_DOWNGRADE*: indicates that a requested feature level downgrade cannot safely be performed

Remove "MinVersionLevel" from ApiVersionsResponse "FinalizedFeatures"

```
    { "name":  "FinalizedFeatures", "type": "[]FinalizedFeatureKey", "ignorable": true,
      "versions":  "3+", "tag": 2, "taggedVersions": "3+",
      "about": "List of cluster-wide finalized features. The information is valid only if
FinalizedFeaturesEpoch >= 0.",
      "fields":  [
        {"name": "Name", "type": "string", "versions":  "3+", "mapKey": true,
          "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", // Drop support for MinVersionLevel in
version 4
          "about": "The cluster-wide finalized min version level for the feature"}
      ]
    }
```

Remove "MinFeatureLevel" from FeatureLevelRecord. Since this record is not yet persisted anywhere in Kafka (KRaft does not yet support feature flags), we can modify this schema without adding a new version.

```
{
  "apiKey": 12,
  "type": "metadata",
  "name": "FeatureLevelRecord",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "Name", "type": "string", "versions": "0+",
      "about": "The feature name." },
    { "name": "FeatureLevel", "type": "int16", "versions": "0+",
      "about": "The current finalized feature level of this feature for the cluster." }
  ]
}
```

Remove the "min_version_level" version from the finalized version range that is written to ZooKeeper (in ZK mode).

KIP-584 CLI Addendum

Re-structure the kafka-features.sh tool to provide functions described by the "Basic" and "Advanced" CLI usages as sub-commands.

```
usage: kafka-features [-h] {describe, upgrade, downgrade, disable}

Options
-------
-h, --help             Show this help message and exit

Commands
--------
describe               Describe one or more feature flags
upgrade                Upgrade one or more feature flags to the given version(s)
downgrade              Downgrade one or more feature flags to the given version(s)
disable                Disable (unset) one or more feature flags
```

All sub-commands require the `--bootstrap-server` argument and may take the `--command-config` argument. All operations that might mutate the state of a feature flag also include a `--dry-run` option.

Describe

The describe command shows the current features active in the cluster.

```
usage: kafka-features --bootstrap-server BOOTSTRAP_SERVER describe [-h]

optional arguments:
  -h, --help             show this help message and exit
```

Upgrade

The ugprade command upgrades one or more features in the cluster.

```
% ./bin/kafka-features.sh upgrade --help
usage: kafka-features --bootstrap-server BOOTSTRAP_SERVER upgrade [-h] [--metadata METADATA] [--feature
FEATURE] [--dry-run]

optional arguments:
  -h, --help             show this help message and exit
  --metadata METADATA    The level to which we should upgrade the metadata. For example, 3.3-IV3.
  --feature FEATURE      A feature upgrade we should perform, in key=value format. For example metadata.
version=3.3-IV3.
  --dry-run              Validate this downgrade, but do not perform it.
```

Downgrade

The downgrade command downgrades one or more features in the cluster.

```
% ./bin/kafka-features.sh downgrade --help
usage: kafka-features --bootstrap-server BOOTSTRAP_SERVER downgrade [-h] [--metadata METADATA] [--feature
FEATURE] [--unsafe] [--dry-run]

optional arguments:
  -h, --help             show this help message and exit
  --metadata METADATA    The level to which we should downgrade the metadata. For example, 3.3-IV0.
  --feature FEATURE      A feature downgrade we should perform, in key=value format. For example metadata.
version=3.3-IV0.
  --unsafe               Perform this downgrade even if it may irreversibly destroy metadata.
  --dry-run              Validate this downgrade, but do not perform it.
```

Disable

The disable command downgrades one or more features in the cluster to level 0 (disabled).

```
% ./bin/kafka-features.sh disable --help
usage: kafka-features --bootstrap-server BOOTSTRAP_SERVER disable [-h] [--feature FEATURE] [--unsafe] [--dry-
run]

optional arguments:
  -h, --help             show this help message and exit
  --feature FEATURE      A feature flag to disable.
  --unsafe               Disable this feature flag even if it may irreversibly destroy metadata.
  --dry-run              Perform a dry-run of this disable operation.
```

KIP-584 Finalized Version Addendum

Remove the concept of a minimum finalized version from feature flags. The supported version will continue to be a min/max range, but the finalized version will simply be the version specified by the operator using kafka-features.sh. This simplifies the KRaft upgrade and downgrade process by eliminating confusion about the meaning of a minimum finalized metadata.version.

New kafka-storage argument

Add `--metadata-version` option to "format" sub-command of kafka-storage.sh. The default value of this initial metadata.version is statically defined by the binary used to run the tool.

```
usage: kafka-storage format [-h] --config CONFIG --cluster-id CLUSTER_ID [--metadata-version VERSION] [--ignore-
formatted]

optional arguments:
  -h, --help             show this help message and exit
  --config CONFIG, -c CONFIG
                         The Kafka configuration file to use.
  --cluster-id CLUSTER_ID, -t CLUSTER_ID
                         The cluster ID to use.
  --metadata-version VERSION
                         The initial value for metadata.version feature flag.
  --ignore-formatted, -g
```

AdminClient changes

To support the three possible downgrade types, we will add an enum and a new constructor to org.apache.kafka.clients.admin.FeatureUpdate

```
public class FeatureUpdate {
    private final short maxVersionLevel;
        private final DowngradeType downgradeType;

        @Deprecated  // Keep this constructor for backwards compatibility
    public FeatureUpdate(final short maxVersionLevel, final boolean allowDowngrade) {
        this(maxVersionLevel, DowngradeType.SAFE);
    }

    public FeatureUpdate(final short maxVersionLevel, final DowngradeType downgradeType) {
        this.maxVersionLevel = maxVersionLevel;
        this.downgradeType = downgradeType;
    }

    public short maxVersionLevel() {
        return maxVersionLevel;
    }

    @Deprecated
    public boolean allowDowngrade() {
        return downgradeType == DowngradeType.SAFE;
    }

    public DowngradeType downgradeType() {
                return downgradeType;
    }

        public enum DowngradeType {
                NONE, SAFE, UNSAFE;
    }
}
```

We will also add a DryRun boolean to UpdateFeaturesOptions with a default no-arg constructor setting the boolean to false.

Proposed Changes

Overview

The sections below go into more detail, but the overall workflow of an upgrade is:

- Operator performs rolling restart of cluster with a new software version

- Operator increases *metadata.version* feature flag using `kafka-features.sh` tool
    - UpdateFeaturesRequest is sent to the active controller
    - The controller validates that the cluster can be upgraded to this version
    - FeatureLevelRecord is written to the metadata log
    - Broker components (e.g., ReplicaManager) reload their state with new version

The downgrade workflow is similar:

- Operator decreases *metadata.version* feature flag using `kafka-features.sh` tool
    - UpdateFeaturesRequest is sent to the active controller
    - The controller validates that the cluster can be safely downgraded to this version (override with --unsafe)
    - FeatureLevelRecord is written to the metadata log
    - Controller generates new snapshot and components reload their state with it (this snapshot may be lossy!)
    - Broker replicates FeatureLevelRecord for downgrade
    - Broker generates new snapshot and components reload their state with it (this snapshot may be lossy!)
- Operator performs rolling restart of cluster with downgraded software version

New Feature Flag

We will introduce a new feature flag named ***metadata.version*** which takes over (and expands on) the role of *inter.broker.protocol.version*. This new feature flag will track changes to the metadata record format and RPCs. Whenever a new record or RPC is introduced, or an incompatible change is made to an existing record or RPC, we will increase this version. The *metadata.version* is free to increase many times between Kafka releases. This is similar to the IV (inter-version) versions of the IBP.

The *metadata.version* feature flag will be defined and configured using the facilities introduced by KIP-584 (feature versions). As brokers and controllers upgrade to new software, their maximum supported *metadata.version* will increase automatically. However, the “finalized” version that can be used by the cluster will only be increased by an operator once all the nodes have upgraded. In other words, the basic workflow of an upgrade is:

- Rolling upgrade software of each node (broker and controller)
- Online upgrade of *metadata.version* to the desired supported version

In the absence of an operator defined value for *metadata.version*, we cannot safely assume anything about which *metadata.version* to use. If we simply assumed the highest supported value, it could lead to unintended downgrades in the event that a broker with a lower supported version joined the cluster. To avoid this, and other upgrade complications, we will need to bootstrap *metadata.version* with some initial version.

Initialization

When the quorum leader (i.e., active controller) is starting up for the first time after this feature flag has been introduced, it will need a way to initialize *metadata.version*. After the controller finishes loading its state from disk, if has not encountered a FeatureLevelRecord, it will read an initial value for this feature from its local *meta.properties* file and generate a FeatureLevelRecord. We will extend the `format` sub-command of `kafka-storage.sh` to allow operators to specify which version is initialized. If no value has been specified by the operator, the tool will select the default value that has been defined for that version of the software.

One special case is when we are upgrading from an existing "preview" KRaft cluster. In this case, the *meta.properties* fill will already exist and will not have an initial *metadata.version* specified. For this case, the controller will automatically initialize metadata.version as 1. In order to maintain compatibility with preview KRaft releases, we must ensure that metadata.version 1 is backwards compatible to KRaft at Apache Kafka 3.0. This allows for a straightforward downgrade path back to an earlier (preview) KRaft software version.

Compatibility

It is possible that brokers and controllers attempt to join the cluster or quorum, but cannot support the current *metadata.version*. For brokers, this is already handled by the controller during registration. If a broker attempts to register with the controller, but the controller determines that the broker cannot support the current set of finalized features (which includes *metadata.version*), the controller will reject the registration request and the broker will remain fenced. For controllers, it is more complicated since we need to allow the quorum to be established in order to allow records to be exchanged and learn about the new *metadata.version*. A controller running old software will join the quorum and begin replicating the metadata log. If this inactive controller encounters a FeatureLevelRecord for *metadata.version* that it cannot support, it should terminate.

In order to ensure that a given *metadata.version* can be used by the quorum, the active controller will check that the given version is compatible with at least a majority of the quorum. Since it's possible for a minority of the quorum to be offline while committing a new metadata.version, we cannot require that all controller nodes support the version (otherwise we affect the availability of upgrades).

In the unlikely event that an active controller encounters an unsupported *metadata.version*, it should resign and terminate.

If a broker encounters an unsupported *metadata.version,* it should unregister itself and terminate.

Upgrades

KRaft upgrades are done in two steps with only a single rolling restart of the cluster required. After all the nodes of the cluster are running the new software version, they will continue using the previous version of RPCs and record formats. Only after increasing the *metadata.version* will these new RPCs and records be used. Since a software upgrade may span across multiple *metadata.version* versions, it should be possible to perform many online upgrades without restarting any nodes. This provides a mechanism for incrementally increasing *metadata.version* to try out new features introduced between the initial software version and the upgraded software version.

One major difference with the static IBP-based upgrade is that the *metadata.version* may be changed arbitrarily at runtime. This means broker and controller components which depend on this version will need to dynamically adjust their state and behavior as the version changes.

ApiVersions

Now that the RPCs in-use by a broker or controller can change at runtime (due to changing *metadata.version*), we will need a way to inform a node's remote clients that new RPCs are available. One example of this would be holding back a new RPC that has corresponding new metadata records. We do not want brokers to start using this RPC until the controller can actually persist the new record type.

Brokers will observe changes to *metadata.version* as they replicate records from the metadata log. If a new *metadata.version* is seen, brokers will renegotiate compatible RPCs with other brokers through the the ApiVersions workflow. This will allow for new RPCs to be put into effect without restarting the brokers. Note that not all broker-to-broker RPCs use ApiVersions negotiation. We will need to likely want to migrate all inter-broker clients to use ApiVersion, but we will evaluate on a case-by-case basis.

Since clients have no visibility to changes in metadata.version, the only mechanism we have for updating the negotiated ApiVersions is connection establishment. By closing the broker side of the connection, clients would be forced to reconnect and receive an updated set of ApiVersions. We may want to investigate alternative approaches to this in a future KIP.

Downgrades

One of the goals of this design is to provide a clear path for *metadata.version* downgrades and software downgrades. Since *metadata.version* can introduce backwards incompatible formats of persisted data, we can classify downgrades into lossless and lossy. If the target downgrade version is fully compatible with the starting version, the downgrade can be executed without any loss of metadata. However, if a new metadata record has been introduced, or an incompatible change was made to a record, a downgrade is only possible if some metadata is removed from the log.

In order to determine if a downgrade can be lossless, developers must indicate if a newly introduced *metadata.version* is backwards compatible or not. This is an implementation detail, but it might look something like:

```
enum MetadataVersions {
  V1(version=1, isBackwardsCompatible=true, description="initial version"),
  V2(version=2, isBackwardsCompatible=true, description="Adding new RPC X"),
  V3(version=3, isBackwardsCompatible=true, description="Adding new optional field to Foo record"),
  V4(version=4, isBackwardsCompatible=false, description="New metadata record type Bar"),
  V5(version=5, isBackwardsCompatible=true, description="New optional field on Bar record")
}
```

In this example, a downgrade from version 5 to 4 would be lossless as would a downgrade from version 3 to any previous version. Only the downgrade from version >=4 to <=3 would be lossy.

When performing a lossless downgrade, no modifications are made to the metadata records. A snapshot is generated which includes the FeatureLevelRecord that downgrades the *metadata.version.* Metadata consumers, including brokers and controllers, will read records at the new (lower) *metadata.version* level. Since a lossless downgrade requires fully backwards compatible changes, the only differences in the record formats can be tagged fields.

If a new record type, or new required fields are added, a lossy downgrade is required. In this case, the snapshot generated by controllers and brokers will *exclude* new record types and will write metadata records at the downgraded version (thereby excluding new fields). By default, the controller should not perform this kind of downgrade since crucial metadata may be lost. The addition of `--unsafe` in the kafka-features.sh tool (and the corresponding `DowngradeType` field value of "2" in UpdateFeaturesRequest) is meant to override this behavior. Using the example above, a lossy downgrade from version 5 to version 2 would mean that "Bar" records would be omitted from the snapshot.

Once the downgrade snapshot has been loaded by all nodes, a software downgrade is now possible. In both lossy and lossless downgrade scenarios, there may be tagged fields present in the metadata records from previous newer version, but these are transparently skipped over during record deserialization.

Metadata Version Change Implementation

Although it falls under the heading of implementation rather than public API, it is worth saying a few words about the implementation of metadata version changes here.

When the MetadataLoader code encounters a metadata version change record:

1. We will write a snapshot to disk at the offset right before the metadata version change.
2. We will serialize the in-memory state to a series of records at the new metadata version.
3. We will publish the in-memory state at the new metadata version.

Importantly, this relies on the property that MetadataDelta only contains changes. So if a specific topic doesn't change during a transition from one metadata version to another, it should not be present in MetadataDelta. This will make performance proportional to what has changed, rather than the overall size of the metadata.

There are several reasons why we prefer to do it this way:

1. In the case of downgrades, it would be quite difficult to maintain correct code for iterating over the entire in-memory state and altering things so that only the data present at a given MV remained. This code would probably end up looking a lot like a metadata image save and load anyway.
2. This tests the implementation of "hot snapshot loading" (that is, loading over an existing snapshot), a code path that otherwise does not receive much testing.

Compatibility, Deprecation, and Migration Plan

Starting with the release that includes this KIP, clusters running self-managed mode will ignore *inter.broker.protocol.version*. Instead, components will begin using *metadata.version* as the gatekeeper for new features, RPCs, and metadata records. The new version will be managed using Kafka's feature flag capabilities.

For clusters in ZooKeeper mode, there may be additional increases of *inter.broker.protocol.version* to introduce new RPCs. While Zookeeper is still supported, will need to take care that whenever an IBP (or IV) is added, a *metadata.version* is also added. Keeping a one to one mapping between IBP and metadata.version will help keep the implementation simple and provide a clear path for migrating ZooKeeper clusters to KRaft.

This design assumes a Kafka cluster in self-managed mode as a starting point. A future KIP will detail the procedure for migrating a ZooKeeper managed Kafka to a self-managed Kafka.

Rejected Alternatives

Transitional feature level

This idea utilizes the fact that KIP-584 feature flags can have a minimum and maximum finalized version. We would consider the cluster to be in a "transitional" version if these two were not equal. For example, min finalized version of 1 and max finalized version of 3. While in this transitional state, the cluster would enable new features introduced (up to the maximum version), but keep the metadata log backwards compatible to the minimum version.

Ultimately, this was determined to be too complex and placed a lot of burden on implementers to figure out various aspects of compatibility. It also increased the burden of knowledge on the operator to understand the implications of this transitional state.

Duplicate records

Similar to the transitional approach above, but in this case the min and max versions of a record would be written to the log together within a generic composite record. For the upgrade scenario, components would only process the newer of the two. For downgrade scenarios, the older (backwards compatible) version would be read. This would allow a downgrade to occur without rewriting the metadata log.

The main downside of this approach is that there is an undetermined amount of time when both versions of a record would be needed in the metadata log. This could lead to increased space requirements for an extended period of time.

Final IBP

Initially, this design included details about a final IBP version that would be used by KRaft clusters to signal the transition from IBP to metadata.version. Instead of this, we decided to simply require that KRaft requires a metadata.version to be set starting with the release that this KIP will be included in. An additional constraint of making version 1 backwards compatible was added to deal with the KRaft preview upgrade case. Overall, this will simplify the code quite a bit