KIP-35 - Retrieving protocol version

Status

Current state: *Accepted*

Discussion thread: here

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

Unable to render Jira issues macro, execution error.

Release: Broker protocol - 0.10.0, Java clients - 0.10.2

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

Motivation

This KIP aims to solve the problem that there is currently no way for a Kafka client to know which API version the broker supports. This means a client might not be able to perform its desired functionality, nor report any meaningful errors back to the application. This makes it hard for clients and applications to support multiple versions of Kafka, which in turn limits the Kafka eco-system since applications and clients will need to be manually built or configured for a specific broker version.

Proposed Changes

In order for a client to successfully talk to a broker, it needs to know what versions of various protocols does the broker support. The KIP suggests the following to achieve that.

1. A new ApiVersionRequest and Response type (version 0) will be added to allow clients to query the broker for supported API request types and versions.

```
ApiVersionRequest => ApiKeys
  // empty

ApiVersionResponse => ApiVersions
  ErrorCode = INT16
  ApiVersions = [ApiVersion]
    ApiVersion = ApiKey MinVersion MaxVersion
      ApiKey = INT16
      MinVersion = INT16
      MaxVersion = INT16
```

2. ApiVersionRequest requests all ApiKeys & versions supported by broker.
3. ApiVersionResponse..MinVersion-MaxVersion semantics:
    a. MinVersion and MaxVersion dictates the lowest and highest supported API versions for the API key (inclusive).
    b. All versions in the MinVersion-MaxVersion must be supported by the broker. Specific version may be deprecated through protocol documentation but must still be supported (broker must not disconnect the client and return a valid protocol response, although it is valid to return an error code if the specific API supports it).
4. ApiVersionResponse.ErrorCode is guaranteed to be the first int16 of the response for all future versions of ApiVersionRequest and is to be used to indicate that the client's ApiVersionRequest with version X (greater than 0) is not supported by the broker and the client should revert to version 0.
5. Clients are recommended to use latest version supported by the broker and itself.
6. Deprecation of a protocol version will be done by marking a protocol version as deprecated in protocol documentation. Documentation shall also be used to indicate a protocol version that must not be used, or for any such information. For instance, say 0.9.0 had protocol versions [0] for api key 1. On trunk, version 1 of the api key was added. Users running off trunk started using version 1 of the api and found out a major bug. To rectify that version 2 of the api is added to trunk. For some reason, it is now deemed important to have version 2 of the api in the next minor release on 0.9.x, i.e., 0.9.1, as well. To do so, version 1 and version 2 both of the api will be backported to the 0.9.1 branch. 0.9.1 broker will return 0 as min supported version for the api and 2 for the max supported version for the api. However, the version 1 should be clearly marked as deprecated on its documentation. It will be client's responsibility to make sure they are not using any such deprecated version to the best knowledge of the client at the time of development (or alternatively by configuration).
   Note that backporting of protocols has not been done in Kafka so far and in practice it can not be done until the Java client is changed so that it doesn't blindly use the latest protocol version. Otherwise, if new request versions were added to 0.9.0.2, the client would not be able to talk to a 0.9.0.1 broker.
7. Supported protocol versions obtained from a broker, is good only for current connection on which that information is obtained. In the event of disconnection, the client should obtain the information from the broker again, as the broker might have upgraded/ downgraded in the mean time.
8. Versions of ApiVersion api should never be deprecated, as clients will rely on the api to inquire supported versions.
9. Java clients will use this API to assert that current API versions are all supported by the brokers it is talking to, and raise an exception if they are not. Java clients check will test the new API.
10. The new API will be tested with librdkafka's implementation as well.
11. The broker returns its full list of supported ApiKeys and versions regardless of current authentication state (e.g., before SASL authentication on an SASL listener, do note that no Kafka protocol requests may take place on a SSL listener before the SSL handshake is finished). If this is considered to leak information about the broker version a workaround is to use SSL with client authentication which is performed at an earlier stage of the connection where the ApiVersionRequest is not available.

Public Interfaces

Addition of ApiVersionRequest and Response version 0.

Compatibility, Deprecation, and Migration Plan

- This is an optional interface and does not affect existing clients.
- Updated clients that add support for the new ApiVersionRequest will be able to adapt their functionality based on supported broker functionality.

Scenarios

A core Kafka developer wants to add a new version of an existing API or a new API

1. Dev adds the new version of API or new API the way they do it today. Protocol documentation, which is auto generated, will automatically be updated with this new version or API in docs. Nothing else required to be done.

A core Kafka developer wants to deprecate a version of an existing API

1. Dev modifies the api description in code to add deprecation information. Protocol documentation, which is auto generated, will automatically be updated with the deprecation information. Note that it will be the client developers responsibility to check deprecation information before using or supporting an api version.

A client developer does not want to be dependent on API versions

1. They continue to operate the way they do without using changes proposed in this KIP.

A client developer wants to add support for a new feature

1. Client sends metadata request and gets to know what brokers it needs to talk to for producing, consuming, etc.
2. Client opens a connection to those brokers and sends ApiVersionRequest.
3. Client collects ApiVersionResponse from all brokers it is interested in and builds a state that will look something like below.
   `ApiVersionResponse received:`

   B1 -> (0, 0, 3), (1, 2, 3)
   B2-> (0, 1, 2), (1, 0, 3), (2, 0, 0)
   Versions state built by client.
   (0, 1, 2) // ApiKey, MinVersion across brokers, Max version across brokers
   (1, 2, 3)
4. Client is expected to have a map of feature to supported ApiVersions. Something like this.
   Feature1 -> {(0, 3, 3), (1, 2, 3)} // ApiKey, MinVersion supported, Max version supported
   Feature2 -> {(0, 0, 1), (1, 2, 3)}
   Note, that this KIP is not proposing to provide this information to clients. It is clients responsibility to maintain this information.
5. Using information from 3 and 4, client can see if it can use a feature or not. For the client in example, the result will be as below.
   Feature1 can not be used as version 3 of api_key 0 is not supported by all interested brokers.
   Feature 2 can be used.

Rejected Alternatives

- Include meta key-value tags in response ("broker.id=...", "message.max.bytes=...", ...)
- Include user-defined tags in response ("aws.zone=...", "rack=...", ..)
- Return array of supported versions, instead of min and max of supported versions, in MetadataResponse.
- Make java clients backwards compatible. This is a much wider discussion and warrants a separate KIP by itself.

Other options that were widely discussed.

Option2: Update Metadata request to carry supported ApiVersions info

Proposed Changes

1. New MetadataRequest and Response version, v1, will be added to provide information on supported protocol versions to clients.

```
MetadataRequest => api_keys, topics
    api_keys => [INT16]
    topics => [STRING]

MetadataResponse => api_versions, brokers, topic_metadata
    api_versions => [single_api_versions]
        single_api_versions => api_key, min_version, max_version
            api_key => INT16
            min_version => INT16
            max_version => INT16
    brokers => node_id host port
        node_id => INT32
        host => STRING
        port => INT32
    topic_metadata => topic_error_code topic [partition_metadata]
        topic_error_code => INT16
        topic => STRING
        partition_metadata => partition_error_code partition_id leader [replicas] [isr]
            partition_error_code => INT16
            partition_id => INT32
            leader => INT32
```

Metadata Request v1 can be used by clients to ask for supported protocol versions by a broker. Client can send Metadata Request, v1, with api_keys set to one of the following values.

   a. Empty array, if client needs supported versions for all ApiKeys.

    b. Array of some ApiKeys, if client needs supported versions for the specified ApiKeys.
    c. Null array, if the client does not need any information on ApiKeys.
2. Broker on receiving a Metadata Request v1, will respond with MetadataResponse v1. The response will contain information on supported versions for api keys specified in the request, if an empty array of api keys is sent, then broker will respond with info on all api keys, and if null array is sent for api keys, then no information on protocol versions is added to the response. Note that broker only provides information on its supported protocol versions.
3. Clients are recommended to use latest api.
4. Clients can choose to not receive topics metadata by specifying a null array for topics in Metadata Request v1.
5. Deprecation of a protocol version will be done by marking a protocol version as deprecated in protocol documentation. Documentation shall also be used to indicate a protocol version that must not be used, or for any such information.For instance, say 0.9.0 had protocol versions [0] for api key 1. On trunk, version 1 of the api key was added. Users running off trunk started using version 1 of the api and found out a major bug. To rectify that version 2 of the api is added to trunk. For some reason, it is now deemed important to have version 2 of the api in 0.9.1 as well. To do so, version 1 and version 2 both of the api will be backported to the 0.9.1 branch. 0.9.1 broker will return 0 as min supported version for the api and 2 for the max supported version for the api. However, the version 1 should be clearly marked as deprecated on its documentation. It will be client's responsibility to make sure they are not using any such deprecated version.
6. Supported protocol versions obtained from a broker, is good only for current connection on which that information is obtained. In the event of disconnection, the client should obtain the information from the broker again, as the broker might have upgraded/ downgraded in the mean time.
7. If a client faces connection closure on sending MetadataRequest Vn, where n is >=1, seeking supported protocol versions, it could be due to following reasons.
    a. Broker does not support version Vn of MetadataRequest. It is advised that clients incrementally try lower versions until a proper response is received.
    b. Connection closure was caused by issues other than unknown MetadataRequest version.

The changes will be tested with librdkafka's implementation.

Scenarios

A core Kafka developer wants to add a new version of an existing API or a new API

1. Dev adds the new version of API or new API the way they do it today. Protocol documentation, which is auto generated, will automatically be updated with this new version or API in docs. Nothing else required to be done.

A core Kafka developer wants to deprecate a version of an existing API

1. Dev modifies the api description in code to add deprecation information. Protocol documentation, which is auto generated, will automatically be updated with the deprecation information. Note that it will be the client developers responsibility to check deprecation information before using or supporting an api version.

A client developer does not want to be dependent on API versions

1. If the clients are using MetadataRequest v0, they can continue operating the way they do without any change in behavior.
2. If the clients are using MetadataRequest >= v1, they can set api_keys as null, and they will not get any info on supported api versions as part of Metadata response.

A client developer wants to add support for a new feature

1. Client sends metadata request and gets to know what brokers it needs to talk to for producing, consuming, etc. and also gets information on the broker's supported api versions. The client maintains this info.
2. Client opens a connection to other brokers that it needs to talk to and sends Metadata request with topics set as null, as it does not want topic metadata again that it has received in previous step.
3. Client collects supported api versions from all brokers it is interested in and builds a state that will look something like below.
   `api versions received:`

   B1 -> (0, 0, 3), (1, 2, 3)
   B2-> (0, 1, 2), (1, 0, 3), (2, 0, 0)
   Versions state built by client.
   (0, 1, 2) // ApiKey, MinVersion across brokers, Max version across brokers
   (1, 2, 3)
4. Client is expected to have a map of feature to supported ApiVersions. Something like this.
   Feature1 -> {(0, 3, 3), (1, 2, 3)} // ApiKey, MinVersion supported, Max version supported
   Feature2 -> {(0, 0, 1), (1, 2, 3)}
   Note, that this KIP is not proposing to provide this information to clients. It is clients responsibility to maintain this information.
5. Using information from 3 and 4, client can see if it can use a feature or not. For the client in example, the result will be as below.
   Feature1 can not be used as version 3 of api_key 0 is not supported by all interested brokers.
   Feature 2 can be used.

Reason for rejection

The only reason for having api_versions info as part of metadata request would have been that it avoids two network cycles. However, in this approach client anyway needs to create connection to each broker to get their supported api versions, so there was no advantage of having api_versions put in metadata response.

Option3: Make protocol documentation as source of truth, instead of code

Proposed Changes

1. Use existing ApiVersion as a unique identifier of protocols' versions supported by a broker. ApiVersion has a monotonically increasing id, which is increased by 1 every time a new version of a protocol is added.

```
sealed trait ApiVersion extends Ordered[ApiVersion] {
  val version: String // Version string, also has information on internal version. E.g., "0.9.0.X", "0.10.0-IV0"
  val id: Int // Monotonically increasing id. E.g., "0.10.0-IV0" has id 4
}
```

2. Maintain protocol documentation per ApiVersion. Each time a new version of any protocol is added, a new ApiVersion, with id as last ApiVersion's id +1, is created and a protocol documentation for the new ApiVersion is created with documentation on newly added version of protocol. The new protocol documentation for the latest ApiVersion will be then added to main protocol documentation page. This procedure on adding protocol documentation for each new ApiVersion will be documented.

   For instance say we have two brokers, BrokerA has ApiVersion 4 and BrokerB has ApiVersion 5. This means we should have protocol documentations for ApiVersions 4 and 5. Say we have the following as protocol documentation for these two versions.

   ```
   Sample Protocol Documentation V4
   Version: 4 // Comes from ApiVersion
   REQ_A_0: ...
   REQ_A_1: ...
   RESP_A_0: ...
   RESP_A_1: ...

   Sample Protocol Documentation V5
   Version: 5 // Comes from ApiVersion
   REQ_A_1: ...
   REQ_A_2: ...
   RESP_A_1: ...
   RESP_A_2: ...
   ```

   All a client needs to know to be able to successfully communicate with a broker is what is the supported ApiVersion of the broker. Say via some mechanism, discussed below, client gets to know that BrokerA has ApiVersion 4 and BrokerB has ApiVersion 5. With that information, and the available protocol documentations for those ApiVersions client can deduce what protocol versions does the broker supports. In this case client will deduce that it can use v0 and v1 of REQ_A and RESP_A while talking to BrokerA, while it can use v1 and v2 of REQ_A and RESP_A while talking to BrokerB.
3. Document requirements on deprecating a version of a protocol. Deprecation of a protocol will be done by marking a protocol version as deprecated in protocol documents corresponding to the ApiVersions in which the protocol version is deprecated. This procedure on deprecating a protocol version will be documented.
4. On receiving a request for an unknown protocol request type, or for an unsupported version of a known type, broker will respond with an empty response only including the common Length+CorrId header with Length=0, instead of closing the connection. The effect of this empty message received by the client is that the client will fail to parse the response (since it will expect a proper non-empty reply) and throw an error to the application with the correct context - that of a protocol parsing failure for request type XYZ. All existing clients should be able to handle this gracefully without any alterations to the code, while updated clients supporting the proposals in this KIP can handle the empty response accordingly by returning a "Request not supported by server" error to the application. The level of detail in the returned error message to the application naturally varies between client implementations but is still by far better than a closed connection which says nothing of the underlying error.
5. New Metadata Request and Response version, v2, will be added to provide ApiVersion information to clients. ApiVersion will be composed of ProtocolVersion, an int, and BuildDescription, a String. BuildDescription should only be used where readability is a concern, like, logs, error-messages, tool outputs, etc.

```
MetadataRequest => Topics
    Topics => NullableArrayOf(STRING)

MetadataResponse => ApiVersion, BrokerEndPoints, TopicsMetaData // ApiVersion is added
    ApiVersion => ProtocolVersion, BuildDescription // E.g., 4, "0.10.0-IV0"
        ProtocolVersion => INT16 // E.g., 4
        BuildDescription => STRING // "0.10.0-IV0", this should only be used for error messages, etc.
    BrokerEndPoints => Same as existing
    TopicsMetaData => Same as existing
```

6. A client willing to and capable of handling multiple protocol versions should send a Metadata Request v2 or above with *topics as null* for *each new connection* to a broker. A broker supporting Metadata Request v2 and above will respond with MetadataResponse that will contain ApiVersion, i.e., ProtocolVersion and BuildDescription, and will *NOT* contain any TopicsMetadata. Note that broker only provides information on its ProtocolVersion and BuildDescription, while responding to a MetadataRequest.
7. Client should get broker's ApiVersion from the broker's MetadataResponse and use the info to determine protocol version it should use to communicate with the broker. Note that ApiVersion obtained for a broker from the broker's MetadataResponse is good only for the current connection. In the event of disconnection, the client should obtain the ApiVersion information from the broker again, as the broker might have upgraded/ downgraded in the mean time.

Scenarios

A core Kafka developer wants to add a new version of an existing API or a new API

1. Dev initiates a KIP, gets approval, gets implementation ready to be committed.
2. Dev claims the next global version id, i.e., current global version id +1.
3. Dev updates the commit with the claimed global version id. The commit should have following pieces.
   a. Code for the new API or new version of an existing API.
   b. Clone of latest protocol documentation for the branch on which changes are being made, updated with information on new version.
   c. Changes to main/ global protocol documentation page with link to the new protocol documentation page created in step b.
4. Changes get committed in a single commit.

A core Kafka developer wants to deprecate a version of an existing API

1. Dev claims the next global version id, i.e., current global version id +1.
2. Dev creates following changes in a commit. The commit should have following pieces.
    a. Clone of latest protocol documentation for the branch on which changes are being made, updated with information on deprecation of concerned api versions.
    b. Changes to main/ global protocol documentation page with link to the new protocol documentation page created in step a.
3. Changes get committed in a single commit.

A client developer does not want to be dependent on API versions

1. They continue to operate the way they do.

A client developer wants to add support for a new feature

1. Client sends metadata request and gets to know what brokers it needs to talk to for producing, consuming, etc. and also gets information on the broker's global api version. The client maintains this info.
2. Client opens a connection to other brokers that it needs to talk to and sends Metadata request with topics set as null, as it does not want topic metadata again that it has received in previous step.
3. Client collects global api versions from all brokers it is interested in and builds a state that will look something like below.
   `api versions received:`

   B1 -> 4
   B2 -> 5
4. Client is expected to have a map of feature to required minimum global api versions. Something like this.
   Feature1 -> 6 // Min. global api version
   Feature2 -> 4
   Note, that this KIP is not proposing to provide this information to clients. It is clients responsibility to maintain this information.
5. Using information from 3 and 4, client can see if it can use a feature or not. For the client in example, the result will be as below.
   Feature1 can not be used as all brokers, client is interested in, do not have >= min. req. global api version.
   Feature2 can be used.

Reason for rejection

This approach makes selective backporting of protocol versions to previous branch difficult, hard to manage. Requires a lot of manual coordination and so has lots of room for mistakes.

Sub-proposal: improved handling of unsupported requests

This proposal was not intended to solve the main feature detection functionality of KIP-35 but rather to fix the suboptimal handling of unsupported requests in the broker to provide meaningful error messages to the end user.

This is an optional nice-to-have feataure and was removed from KIP-35 to minimize scope.

Original text:

> If a request is received for an unknown protocol request type, or for an unsupported version of a known type, the broker should respond with an empty response only including the common Length+CorrId header with Length=0, instead of closing the connection. The effect of this empty message received by the client is that the client will fail to parse the response (since it will expect a proper non-empty reply) and throw an error to the application with the correct context - that of a protocol parsing failure for request type XYZ. All existing clients should be able to handle this gracefully without any alterations to the code, while updated clients supporting the proposals in this KIP can handle the empty response accordingly by returning a "Request not supported by server" error the application. The level of detail in the returned error message to the application naturally varies between client implementations but is still by far better than a closed connection which says nothing of the underlying error.

This proposal is not a good fit for feature discovery since there is no way to reliably and without side-effects check if a certain protocol version is supported. As an example let's say that a future KafkaConsumer needs to commit offsets with OffsetCommit v99 (which features something spiffy), with this proposal the client would need to Join the group, pass through rebalancing, Consume and process message(s) and after that perform an OffsetCommit. If at this point it turns out OffsetCommit v99 wasn't supported the application has processed messages that it now can't commit. This late erroring handling is hard to get right.