KIP-97: Improved Kafka Client RPC Compatibility Policy

Status

Current state: Implemented

Discussion thread: *Here*

JIRA: *KAFKA-4462*

Motivation

Currently, we have a “one-way” backwards compatibility policy. New brokers support older clients, but new clients do not support older broker versions. This policy makes it difficult for users to upgrade Kafka clients. Essentially, they must upgrade to the corresponding broker version before rolling out a new client version.

Upgrading the Kafka client should be a lightweight operation. It should be possible to restart individual clients separately and at different times, since they run in separate processes. However, the one-way compatibility policy forces system administrators to perform a heavyweight broker upgrade before they can execute the lightweight client upgrades.

Currently, users are incentivized to use the oldest client they can put up with. Old clients can talk to both new and old brokers, whereas new clients can only talk to new brokers. Users may miss out on important security, performance, and functionality upgrades because of these perverse incentives. The limited compatibility policy makes deploying and using Kafka more difficult for system administrators.

Proposed Changes

This KIP suggests revising the Kafka Java client compatibility policy so that it is two-way, rather than one-way. Not only should older brokers support newer clients, but newer clients should support older brokers.

Supporting older brokers will require additional code in the client. In some cases, the client will have to maintain two code paths for performing some operation.

In some cases, older brokers will be unable to perform a feature supported by newer clients. For example, retrieving message offsets from a message timestamp (*KIP-79)* is not supported in versions before 0.10.1. In these cases, the new client will throw an *ObsoleteBrokerVersion* exception when attempting to perform the operation. This will let clients know that the reason why the operation cannot be performed is because the broker version in use does not support it. At the same time, clients which do not attempt to use the new operations will proceed as before.

Implementation

Each RPC has a version number which increases monotonically. There are currently about 20 RPCs, with names like Produce, Fetch, JoinGroup, SyncGroup, etc. Currently, the server checks the RPC version of each RPC the client sends, to make sure it can process it. This information is sent over the wire in the beginning of each RPC. The server can handle several versions of client RPCs. But the client always sends the newest version of each RPC that it knows about.

The goal of this KIP is to enable the client to send RPC versions other than the very latest version, so that it can communicate with older brokers. We can use the *ApiVersionRequest* call which was added in *KIP-35.* When it starts up, the client can make a version request call to learn what versions of each RPC the server supports. The client then selects the newest RPC version which the server can understand.

In some cases, the client will need to use an older code path when talking to an older broker. For example, if a new client supports exactly-once semantics, but the broker doesn’t, the client may need to use a different code path to talk to that broker.

In other cases, when the client attempts to do an operation that there is no way to do on the broker, the client will throw an exception locally rather than actually making the RPC. For example, if the client tries to retrieve a message via a timestamp rather than an offset, but the broker is too old to support this feature, *ObsoleteBrokerVersion* will be thrown.

This policy ensures that old software which uses old features will continue to work, while allowing new software to use new features when they become available on the broker.

New or Changed Public Interfaces

No public RPCs will be changed or created. No new public methods will be added to clients. A new public exception, *ObsoleteVersionException*, will be created.

There will be a new command which allows system administrators to find out what RPCs versions all brokers in the cluster support. It will make an *ApiVersionRequest* to each broker, and then print out the results. This may be useful for debugging and troubleshooting version compatibility issues.

The new command will be called BrokerVersionCommand. It will print out broker versions on the command line. The format will show the version range for each broker, as follows:

```
cmccabe@aurora:~/src/kafka> ./bin/kafka-broker-versions.sh --bootstrap-server localhost:9092 --listApiVersions
aurora:9092 (id: 0 rack: null) -> {
 Produce(0): 0 to 2,
 Fetch(1): 0 to 3,
 Offsets(2): 0 to 1,
 Metadata(3): 0 to 2,
 LeaderAndIsr(4): 0,
 StopReplica(5): 0,
 UpdateMetadata(6): 0 to 2,
 ControlledShutdown(7): 1,
 OffsetCommit(8): 0 to 2,
 OffsetFetch(9): 0 to 1,
 GroupCoordinator(10): 0,
 JoinGroup(11): 0 to 1,
 Heartbeat(12): 0,
 LeaveGroup(13): 0,
 SyncGroup(14): 0,
 DescribeGroups(15): 0,
 ListGroups(16): 0,
 SaslHandshake(17): 0,
 ApiVersions(18): 0,
 CreateTopics(19): 0,
 DeleteTopics(20): 0
}
```

Migration Plan and Compatibility

Because we require the ApiVersionRequest RPC to be present, only broker versions beginning with 0.10 will be supported by new clients. However, going forward, new clients will support broker versions after that.

If the burden of backwards compatibility becomes too large, at some point we may need to break it. This is a familiar pattern-- for example, Hadoop maintains backwards compatibility within a single major release, but makes no guarantees across major releases. This decision should be made by the community based on the situation at the time. In general, backwards compatibility is highly useful to users, so this decision should not be made lightly.

Test Plan

The current compatibility tests are:

- tests/kafkatest/tests/upgrade/upgrade_test.py
  - Test upgrade of Kafka broker cluster from 0.8.2, 0.9.0 or 0.10.0 to the current version
- tests/kafkatest/tests/core2/compatibility_test_new_broker_test.py
  - Compatibility tests for moving to a new broker (e.g., 0.10.x) and using a mix of old and new clients (e.g., 0.9.x)
- tests/kafkatest/tests/client1/message_format_change_test.py
  - Tests message format changes

We should add a new test suite which demonstrates that new clients can use older brokers. It should be able to test against multiple older broker versions. In cases where a feature is missing from an older broker, it should check that we receive *ObsoleteBrokerVersion* as expected. In other cases, things should work as expected.

Rejected Alternatives

We considered adding a new RPC to the broker for retrieving “feature flags.” This would return a set of features which the broker supported. The client could then make use of this list of features.

The nice thing about feature flags is that they aren’t tied to a particular RPC. So they can easily describe changes that span multiple RPCs. However, since the feature flag RPC would be a new RPC, it would not be supported in the existing 0.10 and 0.10.1 releases. In contrast, *ApiVersionRequst* already exists in these releases.

We concluded that feature flags are not needed right now, since *ApiVersionRequest* will work pretty well for most use-cases. Most changes to broker functionality can be reasonably described by increasing the version of one or more RPCs. An increase in version doesn't necessarily mean that the data which is sent over the wire is different-- it can also express other semantic changes in how the RPC is designed to be used.

If there are changes that don't make sense in the context of existing RPCs, the *ApiVersionRequest* mechanism allows us to add a brand new RPC. In fact, if we ever decide that feature flags are desirable, we can add them this way.

Potential Future Work

In the future, we might consider adding an API whereby clients can specify which features that they needed when connecting to the broker. This would allow clients to know which features they were getting up-front, rather than learning later. For example, a client could specify that it needs ListOffsetReques tV1 support in order to start up. Or the client could specify that it needs exactly-once semantics in order to start up.

If we ever need to update to a new, incomptible version of the *SaslHandshakeRequest*, we could use this mechanism to do so. The client would simply make the *ApiVersionsRequest* call before setting up SASL. This will probably not be necessary, since the current *SaslHandshakeRequest* is fairly flexible. However, it is an option in the future if we need it.