CLIENTS

Upgrading Apache Kafka Clients Just Got Easier

COLIN MCCABE

JULY 18, 2017

This is a very exciting time to be part of the Apache Kafka® community! Every four months, a new Apache Kafka release brings additional features and improvements. We're particularly excited about the new Streams and Connect features, and exactly-once support. But to get the new stuff, you need to upgrade, and that used to be harder than it is today. In this post, I'll talk a little bit about the new bidirectional client compatibility feature and what it means for users and system administrators.

Though a production deployment of Kafka may support dozens of client applications, none of them need to be stopped in order to upgrade. Rolling upgrades ensure that Kafka is always available during upgrades. Backwards compatibility over the wire ensures that old Kafka clients can talk to new brokers.

The new client compatibility work, introduced in KIP-35 and KIP-97, builds on and extends this backwards compatibility into bidirectional compatibility. Now, you can upgrade the Kafka client software first, rather than upgrading the Kafka brokers first. If you want to try a new version of the Kafka Connect API or Kafka Streams API, just upgrade your client—not your brokers.

Bidirectional Client Compatibility

For a long time, Kafka brokers (the servers that handle messages) have been able to communicate with older versions of the Kafka client software. This "backwards compatibility" enabled rolling upgrades, where Kafka brokers were upgraded while clients continued to run. However, Java clients could not be upgraded until the brokers had been upgraded. Other Confluent source-available clients, like the librdkafka C client, had to be manually reconfigured whenever the broker version changed.

The "bidirectional" client compatibility work done in KIP-35 and KIP-97 removed these limitations. New Java clients can now communicate with old brokers. The Confluent source-available clients for C, C++, and other languages no longer need to be reconfigured when the broker version changes. You can now upgrade the client whenever you want to get access to new features or bug fixes, without worrying about the broker version.

In general, if an operation can be performed by an older broker, we will attempt to perform it—even if it can't be done perfectly. For example, some older brokers don't support certain new request timeout options, but they can still handle the request. In other cases, the client will receive an `UnsupportedVersionException` from the older broker. For example, in the 0.10.1 release, KIP-33 added a time-based log index which can be used to quickly find the messages corresponding to a particular timestamp. Because this does not exist in 0.10.0, a client which calls `KafkaConsumer#offsetsForTimes` to access it will receive an `UnsupportedVersionException`.

Implementation

Implementing bidirectional wire compatibility is an interesting technical problem. How do you write client code to communicate with a broker version that hasn't been designed yet, let alone released?

Kafka brokers understand a fixed set of APIs. For example, `ProduceRequest` is an API that clients use to produce data to Kafka, and `CreateTopicsRequest` is an API that client use to create topics. Each API is identified by a unique number. `ProduceRequest` is number 0; `CreateTopicsRequest` is 19.

APIs also have an associated version number. For example, in the 0.10.0 release, `ProduceRequest` was on version 2, and `CreateTopicsRequest` was on version 0. Whenever an API changes, we increase that version number. Typically, API changes add more fields to the structures sent over the wire, or change the interpretation of existing fields.

Both clients and servers understand multiple different versions of each API. For example, the 0.10.2 broker understands version 3 of `ProduceRequest`, but also versions 0, 1, and 2. It needs to understand the older versions so that it can communicate with older Kafka clients. Similarly, the 0.10.2 client understands both version 1 and 2 of `MetadataRequest`. When it needs to talk to a 0.10.0 broker, it switches to version 1, which the older broker understands. When we switch to an older protocol to facilitate communication with older software, it is called a "protocol downgrade."

So, Kafka has a robust mechanism for uniquely identifying APIs and API versions. The last piece of the puzzle is a mechanism for clients to discover the APIs and API versions a server supports. This is provided by `ApiVersionsRequest`, an API which gives the client a list of all the API and versions supported by the broker. Once the client has this information, it knows how to communicate with the broker.

Of course, there is a bootstrapping problem here. The client has to send the `ApiVersionsRequest` before it knows what versions of `ApiVersionsRequest` the server supports. If the request version is too new, the server will respond with version 0 of `ApiVersionsResponse`, which all clients understand. This is the only case in the Kafka protocol where the version of a response message can be different than the version of the request message that triggered it.

Enabling Improved Client Compatibility

Improved client compatibility is a new feature in Kafka 0.10.2. It is supported by brokers that are at version 0.10.0 or later. From this point forward, all new Kafka clients will feature improved compatibility with older brokers. This will allow administrators to continue running 0.10.0 and 0.10.1 brokers with newer clients for a long time to come.

For the Java client, improved client compatibility is enabled automatically for clients at version 0.10.2 or later. When using another Confluent source-available client such as librdkafka, confluent-kafka-python, confluent-kafka-go, or confluent-kafka-dotnet, you should set the `api.version.request` configuration to `true` to enable improved compatibility. This setting will soon be the default in librdkafka.

Command-line tools

Confluent Platform provides a command-line utility for reading the API versions of brokers, the kafka-broker-api-versions.sh script.

```
cmccabe@aurora:~/confluent-3.2.0/bin> ./kafka-broker-api-versions.sh
Missing required argument "[bootstrap-server]"
Option                                  Description
------                                  -----------
--bootstrap-server <String: server(s)  REQUIRED: The server to connect to.
  to use for bootstrapping>
--command-config <String: command       A property file containing configs to
  config property file>                  be passed to Admin Client.
```

This tool displays the API versions of all the nodes in the cluster. For example, if you have a single broker which is running version 0.10.0, you might see something like this:

```
cmccabe@aurora:~/confluent-3.2.0/bin> ./kafka-broker-api-versions.sh --bootstrap-server localhost:9092
aurora:9092 (id: 0 rack: null) -> (
        Produce(0): 0 to 2 [usable: 2],
        Fetch(1): 0 to 2 [usable: 2],
        Offsets(2): 0 [usable: 0],
        Metadata(3): 0 to 1 [usable: 1],
        LeaderAndIsr(4): 0 [usable: 0],
        StopReplica(5): 0 [usable: 0],
        UpdateMetadata(6): 0 to 2 [usable: 2],
        ControlledShutdown(7): 1 [usable: 1],
        OffsetCommit(8): 0 to 2 [usable: 2],
        OffsetFetch(9): 0 to 1 [usable: 1],
        GroupCoordinator(10): 0 [usable: 0],
        JoinGroup(11): 0 [usable: 0],
        Heartbeat(12): 0 [usable: 0],
        LeaveGroup(13): 0 [usable: 0],
        SyncGroup(14): 0 [usable: 0],
        DescribeGroups(15): 0 [usable: 0],
        ListGroups(16): 0 [usable: 0],
        SaslHandshake(17): 0 [usable: 0],
        ApiVersions(18): 0 [usable: 0],
```

```
        CreateTopics(19): UNSUPPORTED,
        DeleteTopics(20): UNSUPPORTED
)
```


On the left are APIs. On the right are the range of versions which the broker supports. The "usable version" shown in brackets is the highest version which both the broker and the current client software support. Notice that `CreateTopics` and `DeleteTopics` are listed as UNSUPPORTED, since version 0.10.0 of Kafka did not support these APIs.

Conclusion

Kafka's new bidirectional client compatibility decouples broker versions from client versions, and makes upgrades a lot easier. As the Kafka community continues to grow, the new client compatibility policy will provide a solid foundation to build on.