KIP-944: Support async runtimes in consumer

Status

Current state: Withdrawn, because the committers do not seem to be convinced that you cannot control on what thread code runs with an asyn runtime.

Discussion thread: discussion thread, though the discussion was mostly on the vote thread

JIRA: KAFKA-14972

Proposed implementation: pull request 13914

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

Design goal

The goal of this KIP is to allow consumer callbacks to call the consumer again from another thread, while making sure that concurrent access remains impossible.

Motivation

Rebalances cause a lot of message duplication. This can be prevented by doing commits in the partition-revoked callback. This KIP will make it much easier to do work in that callback when an async runtime is used.

The JVM based KafkaConsumer contains a check that rejects nested invocations from different threads (in method acquire). For programs that use an async runtime, this is an almost impossible requirement. Also, the check is more strict than is required; we only need to validate that there is no concurrent access to the consumer.

Examples of affected async runtimes are Kotlin co-routines (see KAFKA-7143) and Zio.

Here follows a condensed example of how we'd like to use ZIO in the rebalance listener callback from the zio-kafka library.

onRevoked callback

```
def onRevoked(revokedTopicPartitions: Set[TopicPartition], consumer: KafkaConsumer) = {
  for {
    _            <- ZIO.logDebug(s"${revokedTps.size} partitions are revoked")
    state        <- currentStateRef.get
    streamsToEnd = state.assignedStreams.filter(control => revokedTps.contains(control.tp)) // Note, we run 1
stream per partition.
    _            <- ZIO.foreachDiscard(streamsToEnd)(_.end(consumer))    // <== Streams will commit not yet
committed offsets
    _            <- awaitCommitsCompleted(consumer).timeout(15.seconds)
    _            <- ZIO.logTrace("onRevoked done")
  } yield ()
}
```

This code is run using the ZIO-runtime as follows from the {{ConsumerRebalanceListener::onPartitionsRevoked}} method:

Running ZIO code from callback

```
def onPartitionsRevoked(partitions: java.util.Collection[TopicPartition]): Unit = {
  Unsafe.unsafe { implicit u =>
    runtime.unsafe
           .run(onRevoked(partitions.asScala.toSet, consumer))
           .getOrThrowFiberFailure()
   ()
  }
}
```

(Note that this code is complex on purpose, starting a ZIO workflow from scratch is not something you would normally do.)

Look at line 6 of the first code block. In method end the stream will try to call `consumer::commitAsync(offsets, callback)`. In `awaitCommitsCompleted()` we call `consumer::commitSync(Collections.emptyMap)` to wait untill all callbacks are invoked.

Since this code is running in the rebalance listener callback, KafkaConsumer enforces that the commit methods must be invoked from the same thread as the thread that invoked `onPartitionsRevoked`. Unfortunately, the ZIO runtime is inherently multi-threaded; tasks can be executed from any thread. There is no way Zio could support this limitation without a major rewrite.

Why can this code not run on a single thread?

We want to use the ZIO runtime. ZIO cannot support this (same argument applies to Cats-effects, a similar and also popular Scala library). To understand why, you first need to know how these libraries work.

In both libraries one creates effects (aka workflows) which are descriptions of a computation. For example, when executing the Scala code `val effect = ZIO.attempt(println("Hello world!"))` one creates only a description; it does not print anything yet. The language to describe these effects is very rich, enough to describe entire applications. Things like concurrency, resource management, timeouts, retries, etc. can all be expressed in an effect. Then to execute the effect, one gives it to the runtime. The runtime then schedules the work on one of the threads in its thread-pool. *Zio, nor Cats-effects supports running an effect on the thread that manages the thread-pool.* Nor is it possible to do so; for example, how would one implement a timeout?

Another reason can be read in 

which talks about Kotlin coroutines. For more information about those: https://kotlinlang.org/docs/coroutines-overview.html

Public Interfaces

Two new methods will be added to `org.apache.kafka.clients.consumer.KafkaConsumer`: `getThreadAccessKey` and `setThreadAccessKey`.

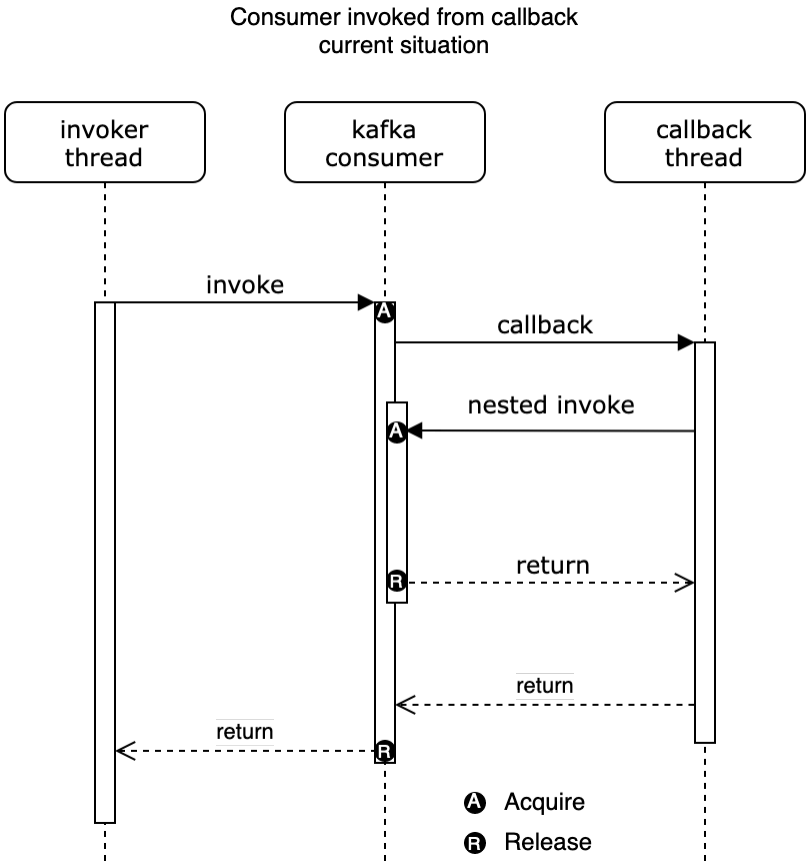
One class is added: `org.apache.kafka.clients.consumer.ThreadAccessKey`.

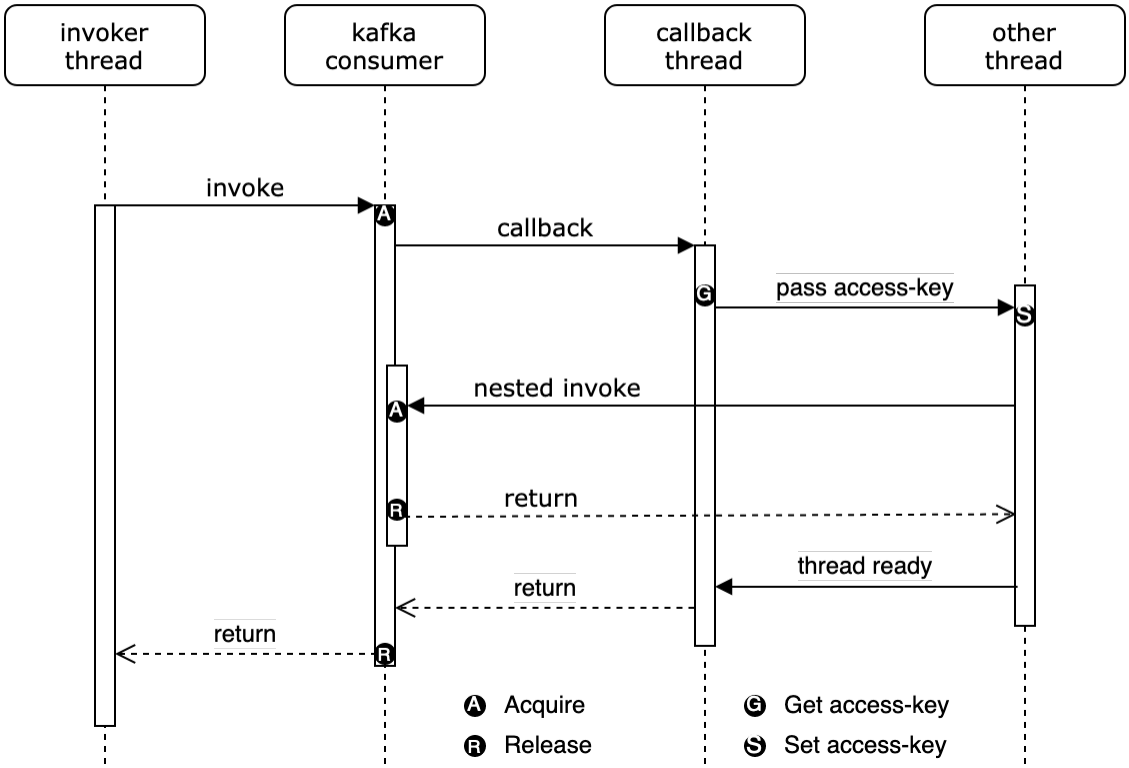
Proposed Changes

In this PR we replace the thread-id check with an access-key that allows a callback to pass on its capability to access the Kafka consumer to another thread.

To keep existing programs working without changes, the access key is stored on a thread-local variable. Developers that work in an async runtime can get the access-key via `getThreadAccessKey` and then activate it on the thread-local variable in a thread of their choosing with `setThreadAccessKey`.

Inside the consumer we maintain a stack of access keys to track which thread is allowed to use the consumer. We need a stack and not a single value because it is possible to have callbacks from callbacks. The top of the stack corresponds to the most recent consumer invocation. An empty stack means that the consumer is not invoked.





Kafka consumer methods that need to be protected against multi-threaded access start with invoking private method `acquire` and end with invoking private method `release`. This KIP changes the implementation of `acquire` and `release`.

When `acquire` is invoked, we first check if access is restricted. It is restricted when the access-key stack is not empty. If it is not empty, the thread-local variable must be equal to the value on the top of the stack. If it is empty, any thread may continue. After this check, we generate a new access-key that can be used inside callbacks. This new access key is pushed on the stack and also stored in the thread-local variable.

When after this, the consumer calls a callback, the callback must be able to invoke the consumer again. This is allowed because the thread-local variable corresponds to the top of the stack. Therefore, code that is not aware of this KIP (all programs in existence till now) will continue to work as before.
The callback may now chose to access the thread-local variable (using `getThreadAccessKey`), and store the access key on the local-variable of another thread (using `setThreadAccessKey`), thereby allowing that thread to access the consumer. Because `acquire` immediately and atomically stores a new access key, it is not possible for multiple threads to use a valid access key concurrently.

When a callback passes its access-key to another thread, it must wait with returning from the callback until that other thread has completed invoking the consumer.

When `release` is invoked, we first validate that the top of the stack is equal to the thread-local variable. If it is not equal, it means that a callback didn't wait for the other thread to complete invoking the consumer. After the check we pop the top value of the access-key stack, and restore the thread-local variable to its previous value. The thread-local variable is restored by copying the new top of the stack into it, or if the stack is now empty we clear the thread-local variable.

Details

We use object identity to compare access keys. For this purpose the class `ThreadAccessKey` is introduced. This has the advantages that it is not possible to guess keys and it gives an efficient implementation.

When one of the described checks in `acquire` or `release` fail, we throw a `ConcurrentModificationException` similar to current behavior of `acquire` and `release`.

Thread safety

Methods `acquire` and `release` need to make sure that memory writes from all threads involved are visible for each other.

The proposed implementation accomplishes this by using a synchronized block on a shared variable. This is sufficient as can be read in the JSR-133 FAQ:

> *But there is more to synchronization than mutual exclusion. Synchronization ensures that memory writes by a thread before or during a synchronized block are made visible in a predictable manner to other threads which synchronize on the same monitor. After we exit a synchronized block, we **release** the monitor, which has the effect of flushing the cache to main memory, so that writes made by this thread can be visible to other threads. Before we can enter a synchronized block, we **acquire** the monitor, which has the effect of invalidating the local processor cache so that variables will be reloaded from main memory. We will then be able to see all of the writes made visible by the previous release.*

For reference, here follows a copy of the proposed implementation of acquire and release.

Class members

```
    // Holds the key that this thread needs to access the consumer, it is used to prevent multi-threaded access.
    private final ThreadLocal<ThreadAccessKey> threadAccessKeyHolder = new ThreadLocal<>();
    // The stack of allowed thread access keys. The top of the stack contains the access key of the thread that
is
    // currently allowed to use the consumer. When the stack is empty, any thread is allowed. Access is
synchronized on
    // the instance.
    private final Deque<ThreadAccessKey> threadAccessStack = new ArrayDeque<>(4);
```

acquire

```
    private void acquire() {
        final ThreadAccessKey threadAccessKey = threadAccessKeyHolder.get();
        final ThreadAccessKey nextKey = new ThreadAccessKey();

        synchronized (threadAccessStack) {
            // Access is granted when threadAccess is empty (consumer is currently not used), or
            // when the top value is the same as current key (consumer is used from callback)
            if (threadAccessStack.isEmpty() || threadAccessStack.getFirst() == threadAccessKey) {
                threadAccessKeyHolder.set(nextKey);
                threadAccessStack.addFirst(nextKey);
            } else {
                final Thread thread = Thread.currentThread();
                throw new ConcurrentModificationException("KafkaConsumer is not safe for multi-threaded access.
" +
                        "currentThread(name: " + thread.getName() + ", id: " + thread.getId() + ")" +
                        " could not provide access key (" + threadAccessStack.getFirst() + ")"
                );
            }
        }
    }
```

release

```
    private void release() {
        final ThreadAccessKey threadAccessKey = threadAccessKeyHolder.get();

        synchronized (threadAccessStack) {
            if (threadAccessStack.isEmpty()) {
                throw new AssertionError("KafkaConsumer invariant violated: `release` invoked without
`acquire`");
            } else if (threadAccessStack.getFirst() == threadAccessKey) {
                threadAccessStack.removeFirst();
                if (threadAccessStack.isEmpty()) {
                    threadAccessKeyHolder.set(null);
                } else {
                    threadAccessKeyHolder.set(threadAccessStack.getFirst());
                }
            } else {
                final Thread thread = Thread.currentThread();
                throw new ConcurrentModificationException("KafkaConsumer is not safe for multi-threaded access.
" +
                        "currentThread(name: " + thread.getName() + ", id: " + thread.getId() + ")" +
                        " returned from callback but not provide access key (" + threadAccessStack.getFirst() +
")"
                );
            }
        }
    }
```

Performance impact of the synchronized block is minimal because there will be no contention. Contention can only be caused by a badly written client and always results in a `ConcurrentModificationException`.

Compatibility, Deprecation, and Migration Plan

For existing users nothing changes, only the exception message for using the consumer from the wrong thread changes.

There is no need to deprecate anything. No migration is needed.

Test Plan

Unit tests are sufficient. The first step is to find or write tests that test the current thread-id based locking. These test must continue to work with the proposed locking. The next step is to add more unit tests to verify the new behavior.

Unit tests to test thread-id based locking:

- in callback, invoking consumer from the same thread is allowed
- in callback, invoking consumer from a different thread is rejected

Additional unit tests:

- in callback, invoking consumer from a different thread is allowed when access key is provided
- in callback, invoking consumer concurrently from multiple threads is rejected even when access key is provided

Rejected Alternatives

Alternative A: add a configuration to disable the thread-id check

Disabling the thread-id check based on configuration would be a very easy change. However, without the check it will become very easy to use the consumer wrong, especially from multi-threaded asynchronous runtimes.

Alternative B: disallow concurrent invocations, but allow them from any thread

This is a stronger approach than alternative A, but still a lot weaker than the proposed change. For example, with this alternative, when a callback is running, a completely unrelated thread may use the consumer. Since that thread is unrelated there is no coordination between when the callback ends and the other thread causing the consumer to be running on multiple threads after all. This can lead to very hard to track bugs.