Consumer threading refactor design

Summary

This document highlights our effort to refactor the current KafkaConsumer. This project aims to address issues and shortcomings we've encountered in the past years, such as increasing code complexity and readability, concurrency bugs, rebalancing issues, and, lastly, the enable KIP-848.

- **Code complexity**: Patches and hotfixes in the past years have heavily impacted the readability of the code. The complex code path and intertwined logic make the code difficult to modify and comprehend. For example, coordinator communication can happen in both the heartbeat thread and the polling thread, which makes it challenging to diagnose the source of the issue when there's a race condition, for example. Also, many parts of the coordinator code have been refactored to execute asynchronously; therefore, we can take the opportunity to refactor the existing loops and timers.
- **Complex Coordinator Communication**: Coordinator communication happens at both threads in the current design, and it has caused race conditions such as KAFKA-13563. We want to take the opportunity to move all coordinator communication to the background thread and adopt a more linear design.
- **Asynchronous Background Thread**: One of our goals here is to make rebalance process happen asynchronously with the poll. Firstly, it simplifies the design and logic because the timing of the network request is more predictable. Secondly, because rebalance occurs in the background thread, the polling thread won't be blocked or blocking the rebalancing process.

Scope

- Define the responsibilities of the polling thread and the background thread
- Define the communication interface and protocol between the two threads
- Define a Network layer over the NetworkClient, including how different requests are sent and handled
  - Define RequestManagers
  - Define RequestStates to handle retry backoff
- Redesign SubscriptionState
- Deprecate HeartbeatThread and move it to the background thread
- Define rebalance flow, including how JoinGroup and SyncGroup request is sent and how callbacks are triggered
- Refactor the KafkaConsumer API innards
- Address issues in these Jira tickets

Design

The new consumer is asynchronous and event drive. Former means we are separating the responsibility between the polling thread and the background thread. The latter means the two threads communicate via events. The core components are:

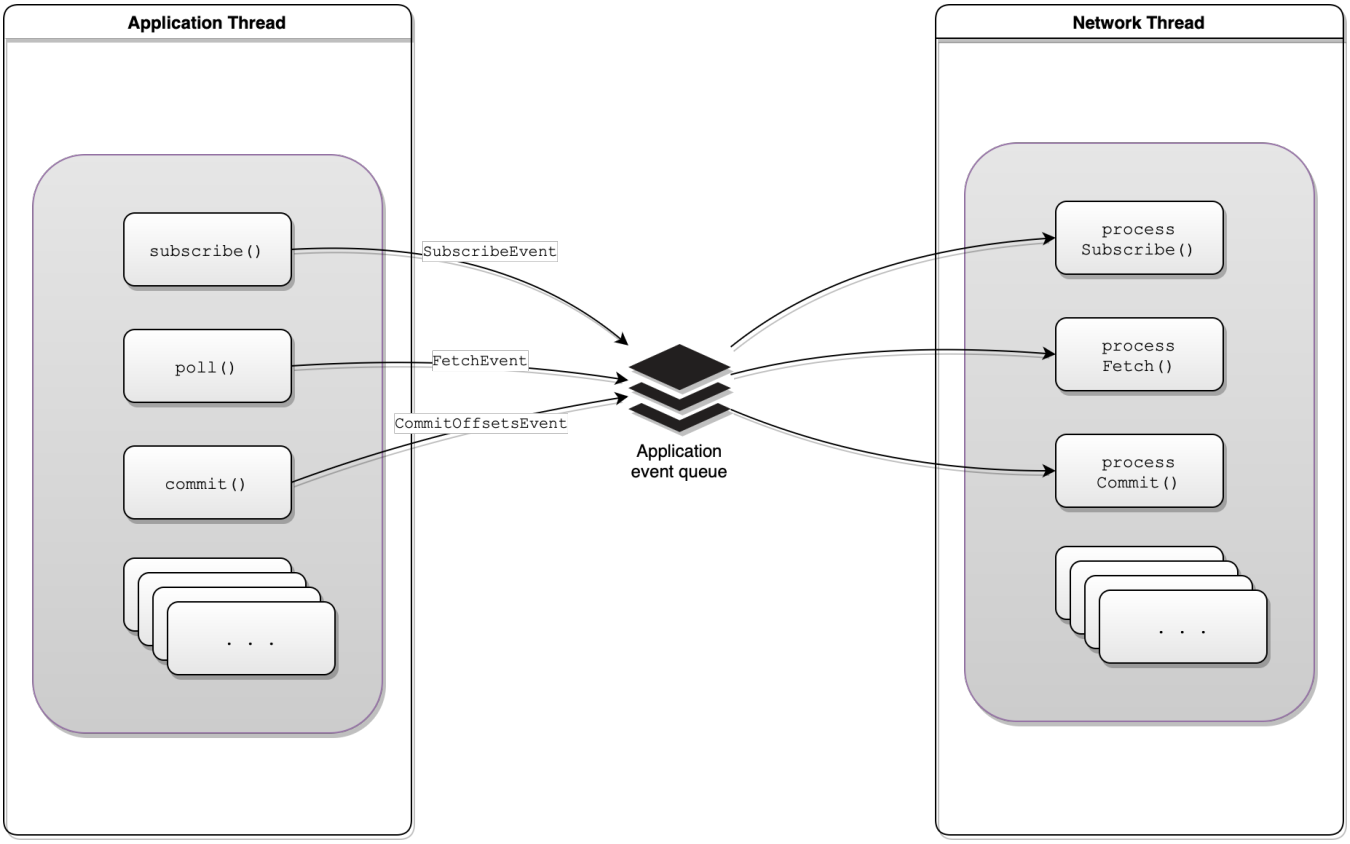
- **Application thread:** user thread on which all `Consumer` API requests and callback handlers are executed.
- **Network thread:** client-internal thread on which network I/O related to heartbeat, coordinator, and rebalance is executed.
- **Event handler**: A communication interface between the application thread and network thread

- **Application event queue***:* sends events to the network I/O thread from the application thread
- **Background event queue***:* sends events from the network thread to the application thread
- **Application event processor**:

We will discuss the strategy for handling event results in the following section. In short, for async APIs like commitAsync(), *CompletableFuture* will be used to notify the completion of the events.

Application and Network Thread

The following diagram depicts the interaction between the application thread and the network thread:



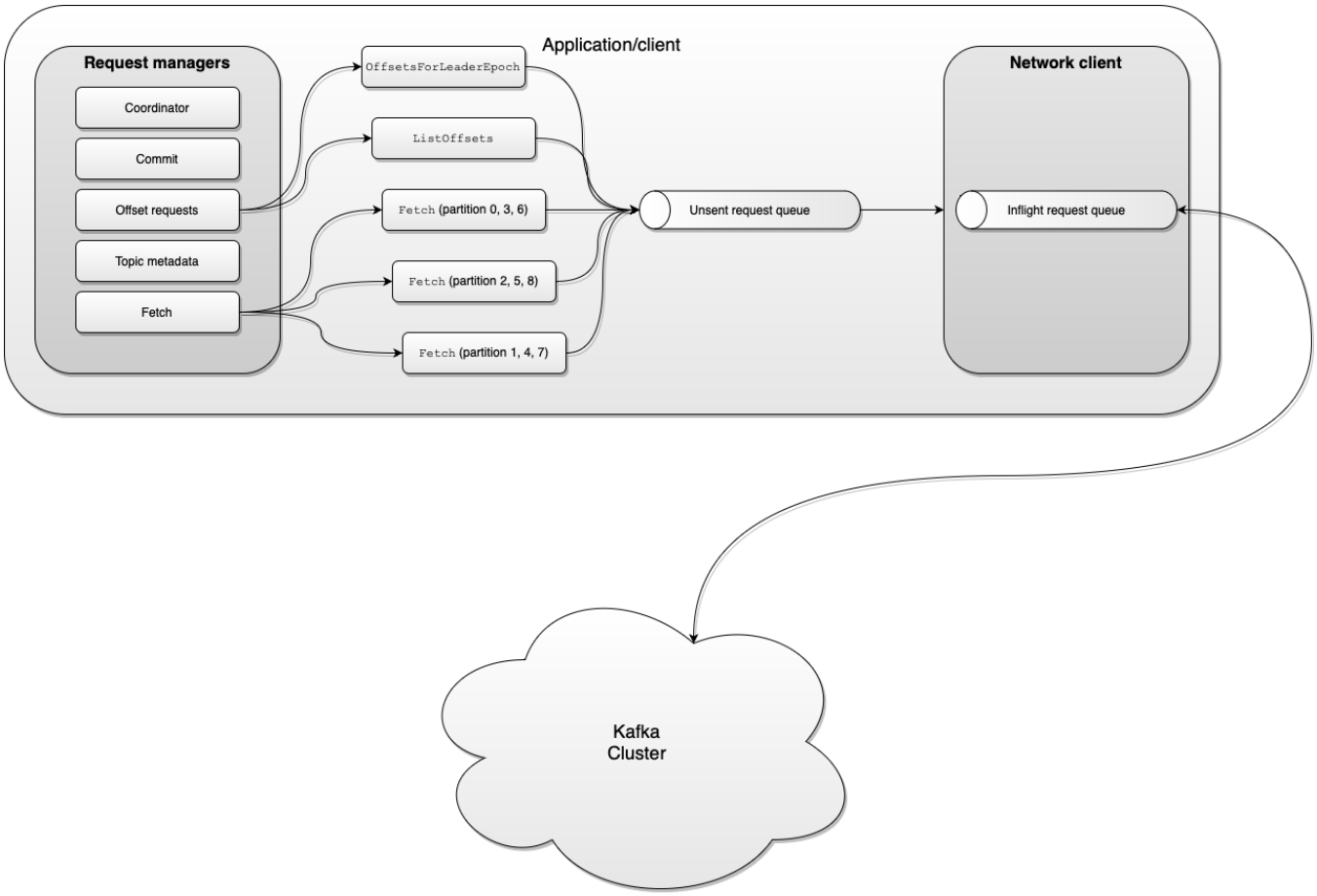
The `Consumer` client object is here depicted in purple. In this design, instead of directly operating on the parameters given to the various APIs (`subscribe()`, `poll()`, `commit()`, etc.), the `Consumer` implementation packages the parameters as events that are enqueued on the application event queue.

The event handler, shown in green, executes on the network thread. In each loop of the network thread, events are read from the queue and processed.

If the `Consumer` API is a blocking call, the event passed from the application thread to the network thread will include an embedded `CompleteableFutur`. After enqueuing the event, the application thread will invoke `Future.get()`, effectively blocking itself until a result is provided. When the result for the `Consumer` API call is ready, the network thread will then invoke `CompeteableFuture.complete()` with the result, allowing the application thread to continue execution.

Submitting Client Requests

The following diagram displays the basic flow between the request managers, unsent request queue, and the `NetworkClient`:

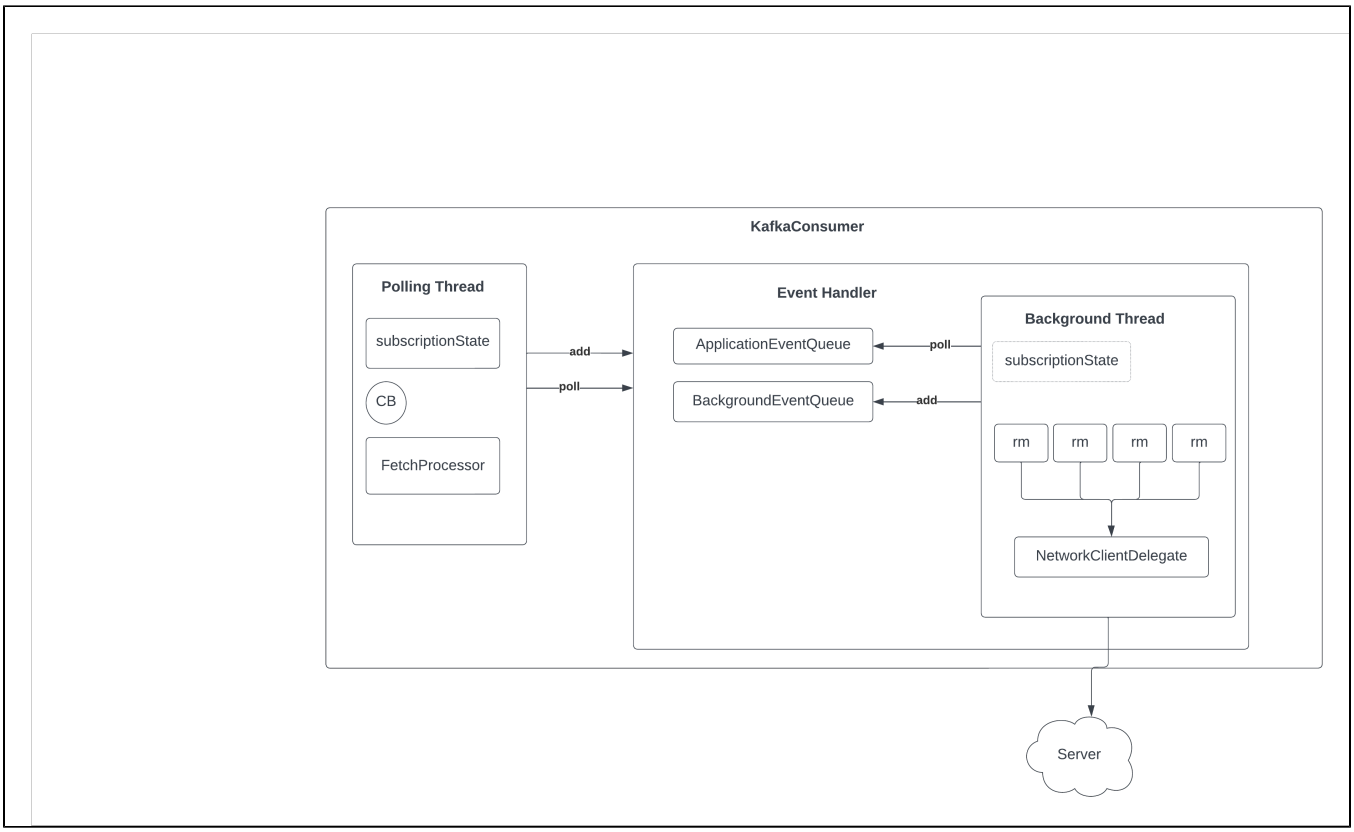


A request manager represents the logic and stated needed to issue a Kafka RPC requests and handle its response. A request manager may contain logic to handle more than one type of RPC.

In the network thread, we loop over each request manager, effectively asking it for any requests that it needs to send to the cluster. Note that during this step the network request is not sent. Instead, an *unsent request* object is created which contains the underlying request information. These "unsent requests" are added to a queue of pending unsent client requests. After all of these unsent requests are queued up, then they are forwarded for network I/O via the `NetworkClient.send()` method.

There are two benefits for this multi-step process:

1. It keeps the network I/O request and response management and lifecycle in one place, making the code easier to reason about
2. The request managers can implement deduplication and/or coalescing of requests



Terminologies:

- CB: Callbacks registered and invoked on the polling thread: commit callback, rebalance callback.
- rm: RequestManagers. e.g. Heartbeat, FindCoordinatorRequest.
- subscriptionState design is still under discussion

Application thread and its lifecycle

The polling thread handles API invocation and any responses from the background thread. Let's demonstrate its life cycle using the simple consumer use case (assign(), poll(), commitSync()) :

1. The user invokes assign(), the subscriptionState is altered.
2. The subscription state changes are sent to the background thread via the BackgroundEventQueue.
3. The user then invokes poll() in a loop.
4. During the poll, the polling thread sends a fetch request to the background thread.
5. During the poll, the polling thread polls fetch results from the BackgroundEventQueue. It deserializes the poll results and returns the result to the user.
6. The user processes the results and invokes commitSync().
7. The client thread sends an OffsetCommitApplicationEvent to the background thread. As this is a blocking operation, the method returns when the background thread completes the commit.

Background thread and its lifecycle

The background runs a loop that periodically checks the ApplicationEventQueue, and drains and processes the events. On the high level, the lifecycle of the background thread can be summarized as such:

1. The application starts up the Consumer, the Consumer creates an EventHandler, and starts up the background thread.
2. The background thread enters the loop and starts polling the ApplicationEventQueue.
    a. Events will be sent to the corresponding RequestManager. For example, a commit event is sent to the OffsetCommitRequestManager.
3. The background thread polls each RequestManager. If the RequestManager returns a result, we enqueue it to the NetworkClientDelegate.
4. Poll the NetworkClientDelegate to ensure the requests are sent.

Network Layers

blocked URL

We are deprecating the current ConsumerNetworkClient because:

1. The lockings are unnecessary in the new design because everything is on a single thread.
2. Some irrelevant features are irrelevant to this design, such as unsent.

We are introducing a wrapper over NetworkClient, the NetworkClientDelegate, to help to coordinate the requests.

- All requests are first enqueued into the *unsentRequests* queue
- Polling the NetworkClient will result in sending the requests to the queue.

Request Manager

Kafka consumer tasks are tight to the broker requests and responses. In the new implementation, we took a more modular approach to create request managers for different tasks and have the background thread to poll these request managers to see if any requests need to be send. Once a request is returned by the poll, the background thread will enqueu it to the network client to be sent out.

The request managers handle the following requests

1. FindCoordinatorRequest
2. OffsetCommitRequest
3. FetchRequest
4. MetadataRequest
5. HeartbeatRequest
6. ListOffsetRequest

After KIP-848 is implemented, the request managers also handle the following:

1. ConsumerGroupHeartbeatRequest
2. ConsumerGroupPrepareAssignmentRequest
3. ConsumerGroupInstallAssignmentRequest

RequestFuture and Callback

The current implementation chains callbacks to requestFutures (Kafka internal type). We have decided to move away from the Kafka internal type and migrate to the Java CompletableFuture due to its better interface and features.

Events and EventHandler

EventHandler is the main interface between the polling thread and the background thread. It has two main purposes:

1. Allows polling thread to send events to the background thread
2. Allows polling thread to poll background thread events

Here we define two types of events:

1. ApplicationEvent: application side events that will be sent to the background thread
2. BackgroundEvent: background thread events that will be sent to the application

We use a blocking queue to send API events from the polling thread to the background thread. We will abstract the communication operation using an EventHandler, which allows the caller, i.e. the polling thread, to add and poll the events.

EventHandler

```
interface EventHandler {
  public ApplicationEvent poll();
  public void add(RequestEvent event);
}
```

ApplicationEventQueue and ApplicationEvent

```
// Channel used to send events to the background thread

private BlockingQueue<ApplicationEvent> queue;

abstract public class ApplicationEvent {
   private final ApplicationEventType eventType;
}

enum ApplicationEventType {
   COMMIT,
   ACK_PARTITION_REVOKED,
   ACK_PARTITION_ASSIGNED,
   UPDATE_METADATA,
   LEAVE_GROUP,
}
```

BackgroundEventQueue and BackgroundEvent

```
// Channel used to send events to the polling thread for client side execution/notification

private BlockingQueue<BackgroundEvent> queue;

abstract public class BackgroundEvent {
   private final BackgroundEventType eventType;
}

enum BackgroundEventType {
   ERROR,
   REVOKE_PARTITIONS,
   ASSIGN_PARTITIONS,
   FETCH_RESPONSE,
}
```

Consumer API Internal Changes

Poll

The users are required to invoke poll to:

1. Trigger auto-commit
2. Poll exceptions: process or raise it to the user
3. Poll fetches
4. Poll callback invocation trigger to trigger the rebalance listeners.

CommitSync

1. The polling thread send a commit event. The commit event has a completable future.
2. Wait for the completable future to finish, so that we can make this a blocking API

Assign

1. If we are assigning nothing, trigger unsubscribe()
2. clear the fetcher buffer
3. send a commit event if autocommit is enabled
4. send metadata update

Subscribe

1. If subscribing to nothing, trigger unsubscribe()
2. clear the fetcher buffer
3. subscribes
4. send metadata update

Unsubscribe

1. Send a leave group event
2. unsubscribe from the topics

Major Changes

Fetcher

We will break the current fetcher into three parts to accommodate the asynchronous design, i.e., we need the background thread to send fetches autonomously and the polling thread to collect fetches when these fetches become available. We will have 3 separate classes here:

1. FetchSender: Responsible for sending fetches in the background thread
2. FetchHandler: Sitting in the polling thread's poll loop, processing the fetch response from the fetch event.
3. FetchBuffer: This is the CompletedFetches in the old implementation. The use case prevents the FetchSender from sending too many fetches and causing memory issues. This will be removed once we implement the memory-based buffer.(KIP-81)

Consumer Poll Changes

We will remove the metadata logic from the consumer.poll(), so that the execution of the poll loop is much simplified. It mainly handles:

1. fetches
2. callback invocation
3. errors

ConsumerCoordinator and AbstractCoordinator

- New states will be introduced (see Rebalance States section above). The main purpose is to make the background thread drive the poll, and letting the polling thread to invoke the callbacks.
- Remove HeartbeatThread. Therefore, we won't be starting the heartbeat thread.
    - It will still require a fetch event to poll heartbeat. As only polling thread issues fetch events, and we want to respect the existing implementation.
- Timeouts and timers will be reevaluated and possibly removed.
- while loops will be reevaluated and possibly thrown out. In the new implementation the coordinator will be non-blocking, and its states are managed by the background thread loop.

Timeout Policy

Please see Java client Consumer timeouts for more detail on timeouts.

Compatibility

The new consumer should be backward compatible.

Alternative Proposals

Fetcher

- Should some of the fetcher configuration be dynamic
- Configurable prefetch buffer

SubscriptionState

- Non-shared: We can make a copy of the subscriptionState in the background thread, and use event to drive the synchronization.
    - There could be out of sync issues, which can subsequently causes in correct fetching, etc..

API Changes

- Poll returns CompletableFuture<ConsumerRecord<K,V>>

User Adoption

The refactor should have (almost) no regression or breaking changes upon merge. So user should be able to continue using the new client.