# Java client Consumer timeouts



## Overview

Many of the `Consumer` APIs provide a means for users to express that an operation should adhere to a timeout. This is achieved by including a `Duration` object as one of the API method parameters.

Take as an example `commitSync()` and `commitSync(Duration timeout)`:

```
public void commitSync()

public void commitSync(Duration timeout);
```

Not all of the APIs support timeouts, but of those that do, the timeout is either *required* or *optional*.

The following `Consumer` APIs *require* a timeout:

- `clientInstanceId`
- `poll`

The following `Consumer` APIs provide overloaded versions that allow the user to pass in an *optional* timeout:

- `beginningOffsets`
- `close`
- `commitSync`
- `committed`
- `endOffsets`
- `listTopics`
- `offsetsForTimes`
- `partitionsFor`
- `position`

Consumer.poll() - user provide timeout

Coordinator rediscovery backoff: retry.backoff.ms

Coordinator discovery timeout: Currently uses the user-provided timeout in the consumer.poll(). Maybe we should use request.timeout.ms. And re-attempt in the next loop if failed

CommitOffsetSync: user provided

Rebalance State Timeout: maybe using the request timeout

*Is there a better way to configure session interval and heartbeat interval?*

### Interpretation of Timeout Values

A precise definition of the timeout policy of the existing `Consumer` is undefined. The main clues as to the intended behavior is based on the API-level documentation as well as the source code itself. The documentation can be a little vague and the source code is not consistent throughout the different API implementations. Also, Kafka does not provide any real time guarantees, so the level of precision in describing the timeouts is rough. This leaves us in the situation in which there may be more than one way to interpret how a timeout is implemented.

## Types of Timeouts

There is no one *type* of a timeout; there are many, which can cause confusion. We will focus on just two types of timeouts: API timeouts and network I/O timeouts.

### API Timeouts

As an example, let's imagine a user has developed their Kafka application such that a value of `10,000` is passed in to the `poll()` method. Intuitively, what does the user expect the behavior to be? The user would expect that client would do its best to return as many records as it can within that limit of 10 seconds. The application would invoke the Kafka client, and for up to 10 seconds the application may be waiting for a response.

In this document, we refer to the timeout that the user supplies to an API as the *API timeout*. This is a timeout that covers the entirety time spent on the Consumer API call. The user should be free to treat the API timeout like a black box; it is the upper-bound on the length of time spent executing that API call. When a timeout-based `Consumer` API is invoked, that timeout value provides an upper-bound for the aggregation of the entire set of operations required by that API call. That is, the length of time for all the constituent operations of that API call must be less than or equal to the timeout provided by the user.

In the overview, we stated that the `Consumer` APIs provides overloaded versions of many methods with an *optional* timeout. Say the user calls `commitSync()` (i.e. the version of the commit method that does not include a timeout)—is it then assumed that the method will run forever? The documentation for `default.api.timeout.ms` states this about the configuration option:

> *Specifies the timeout (in milliseconds) for client APIs. This configuration is used as the default timeout for all client operations that do not specify a `timeout` parameter.*

So the implementation of method such as `commitSync()` essentially just calls its sibling version (`commitSync(Duration timeout)`) like this:

```
public void commitSync() {
    Duration timeout = Duration.ofMillis(defaultApiTimeoutMs);
    commitSync(timeout);
}
```

## Network I/O Timeouts

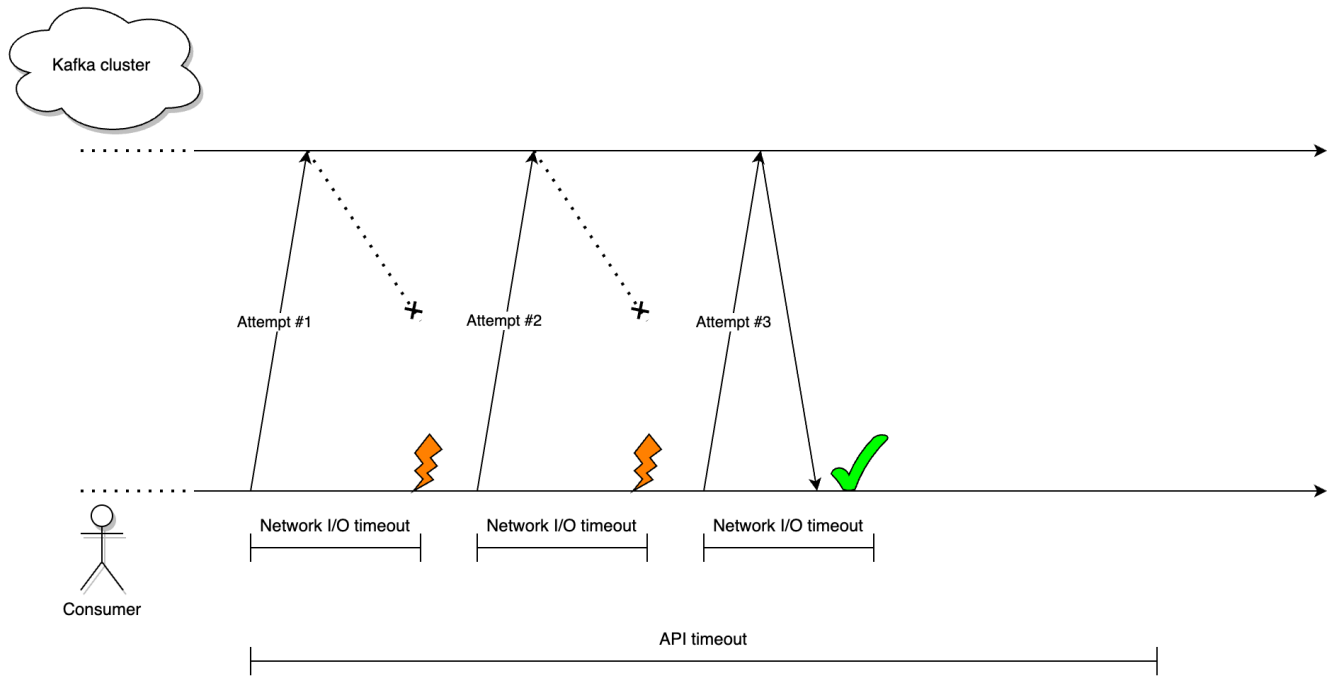
In practice, timeouts are largely used to time-bound I/O. In the case of a Kafka client, there is no disk I/O, so we can focus our attention solely on network I/O. The communication between the client and brokers over the network is going to constitute the bulk of the time for many operations. Allowing the user to provide an upper bound on the total time of these operations provides some protection against network issues.

For API calls that require network I/O operations, the `Consumer` will issue network requests to the Kafka cluster. Each of those distinct network requests include their own timeout value, which we refer to as the *network I/O timeout*. Network I/O timeouts are *not* provided directly as part of the `Consumer` API. Instead, they are supplied to the client at initialization time via the `request.timeout.ms` configuration option.

## Relationship Between API Timeouts and Network I/O Timeouts

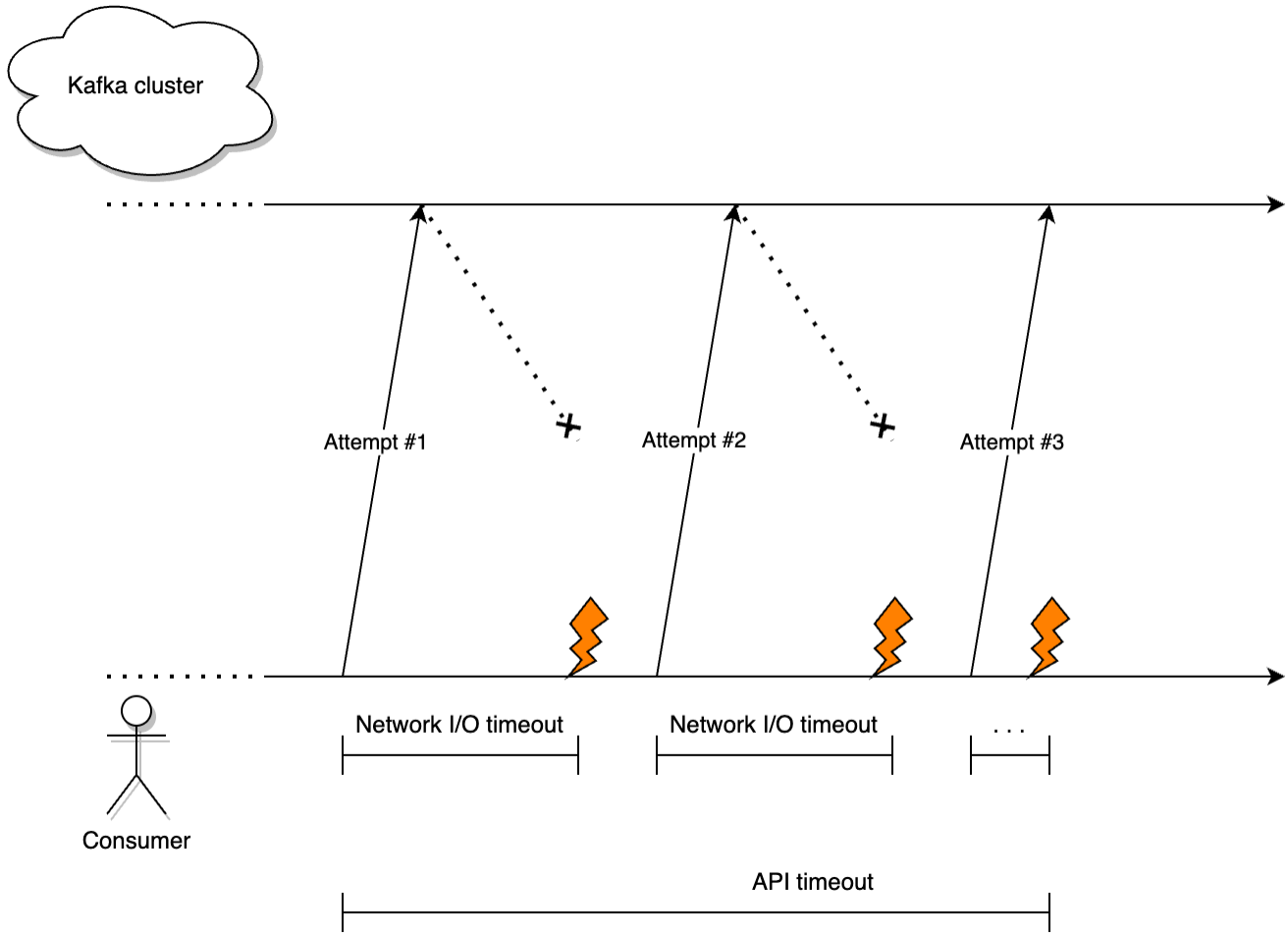
Let's look at a couple of examples to highlight the difference between these two timeouts.

In our first diagram, the user has invoked a `Consumer` API call with a very generous timeout:



In this first example, the API call needs to make a network I/O request to the Kafka cluster. Unfortunately, from the client's perspective, something is wrong with the network and/or the broker because it did not receive a response within the configured network I/O timeout. Because there was sufficient time left within the API timeout, the client was able to retry the network request until it succeeded on the third attempt. This retry functionality is implemented within each `RequestManager` implementation as not all requests should be retried in all cases.

In our second diagram, the user has invoked a `Consumer` API call with a much shorter timeout:




As in the first example, when the client attempts to make a network I/O request to the Kafka cluster, it is not receiving the response within the configured network I/O timeout. Because there was sufficient time left within the API timeout, the client was able to retry the network request. However, notice the third network I/O timeout is much shorter than the previous two. Why is that? As mentioned above, normally the network I/O timeout would be determined by the `request.timeout.ms` configuration. However, in order to ensure the client abides by the overall *API timeout*, we must reduce the network I/O timeout of the third request. Thus we must always make sure

```
int requestTimeoutMs = Math.min(apiTimeoutRemainingMs, requestTimeoutMs);
```

# Timer

When a user provides a timeout value to a `Consumer` API, a `Timer` object is immediately created to track the elapsed/remaining time for processing. While a `Duration` object provides a fixed value of the overall timeout, the `Timer` tracks how much time remains since it was first created. At certain points during processing, the `Timer.update()` API is invoked to determine the elapsed/remaining time for processing.

The logic in the `Timer` class does not in itself magically enforce any timeouts. The code that uses the `Timer` object must interact with it explicitly to update it (`update()`) and query it (`remainingMs()`, `isExpired()`, and not`Expired()`) to determine the remaining value of the timeout.

The `Timer` class is not designed to be thread-safe. Although it might be useful to reuse the same `Timer` object in both the application and network I/O threads, this is currently ill-advised due to the lack of thread safety. This will likely force us to have two separate `Timer` instances (one for each thread), which is less than ideal 😞