Proposal for Google Summer of Code 2023

Dataplane migration for Apache Kafka communications: Leveraging Java19's Project Loom

1. Project Details

- Title

  Dataplane migration for Apache Kafka communications: Leveraging Java19's Project Loom

- Project Overview

  Knative eventing is a system designed to meet a frequent demand for cloud-native development. It offers composable primitives that enable late-binding between event sources and event consumers. The Knative Eventing system's **Eventing Kafka Broker** is the part that offers a mechanism to consume and output events using Apache Kafka as the underlying messaging infrastructure.

  In the current scenario, the Knative Eventing Kafka Broker's data-plane communication with Apache Kafka for consuming and producing records is done via the Vert.x-Kafka-client library which is a wrapper for communications with Apache Kafka inside the Vert.x threading model.

- Objective

  This project idea aims to implement the Knative Kafka Broker data-plane communication with the native Apache-kafka-client library working on Java 19 and evaluate OpenJDK 19's Project Loom and leverage its virtual threads for efficient and concurrent communication with the Apache Kafka cluster. [eventing-kafka-broker/Issue #2928]

2. Deliverables

A. Implement a new reactive minimal interface `KafkaProducer` and `KafkaConsumer` object implemented over the native Apache Kafka client library with Virtual Threads and Vert.x-Kafka-client library.
B. Testing of the implementation, including Unit tests and report detailing the test results and any issues encountered during the testing process.
C. Comparison of the two implementations' performance, flexibility, and caveats (Vert.x-Kafka-client library and virtual threads with native Apache Kafka client library).
D. Documentation of the implementation, and a discussion of the results and implications of the performance comparison.

3. Implementation Details

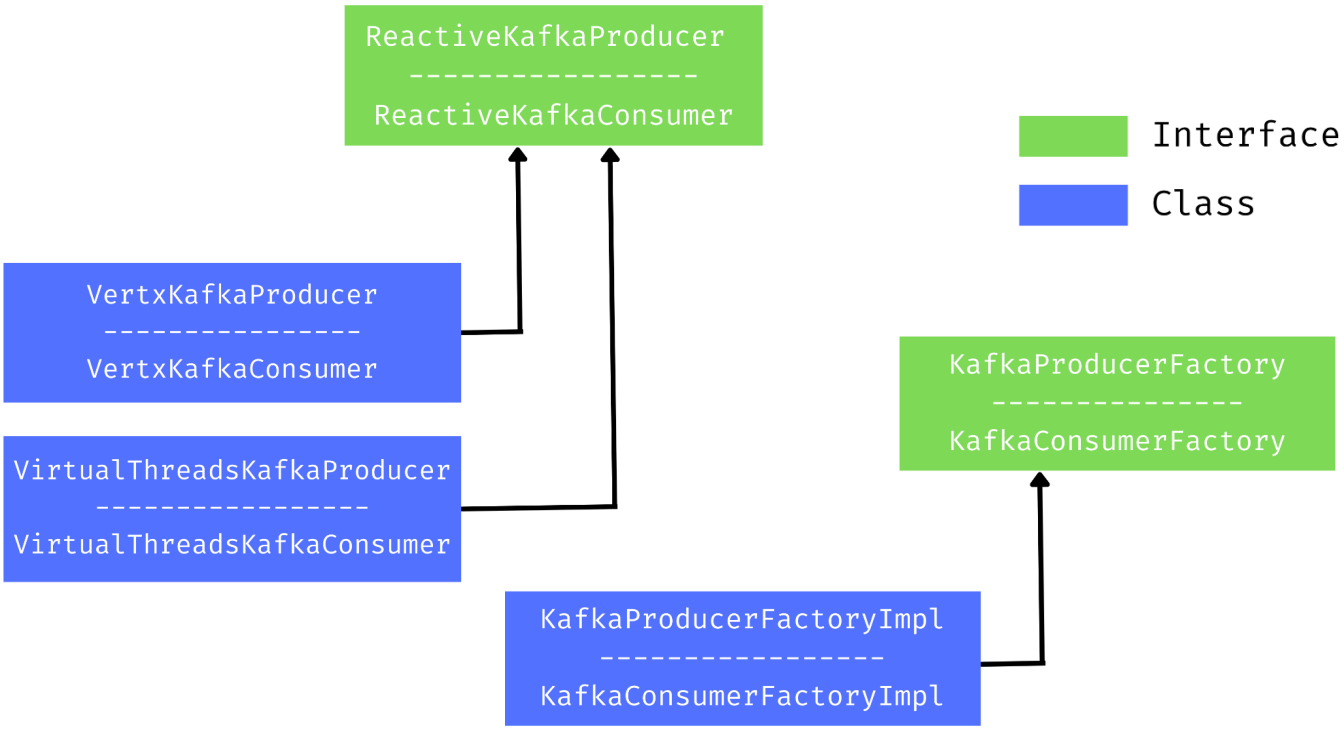
There are two main components -

1. Receiver (Acts as Kafka Producers)
2. Dispatcher (Acts as Kafka Consumer)

For both of them we create our own reactive minimal interfaces for KafkaProducer and KafkaConsumer and then implement those twice:

- one with the existing Vert.x-Kafka-Client library
- one with Virtual Threads + Raw Apache Kafka client library.

The class diagram would be like below:



- Receiver
- The new custom `ReactiveKafkaProducer` interface consists of all the methods signatures we use in the Receiver component.

```
interface ReactiveKafkaProducer {
   Future<RecordMetadata> send(ProducerRecord<K,V> record);
   Future<Void> close();
   Future<Void> flush();
   KafkaProducer<K,V> wnwrap();
   // ... other producer method signatures here
}
```

- `VertxKafkaProducer class` will implement `ReactiveKafkaProducer` using Vertx-Kafka-Client library. For example, the `send()` method may look:

```
Future<RecordMetadata> send(ProducerRecord<K,V> record) {
   return this.producer.send(
     KafkaProducerRecord.create(record.topic(),record.value())
   );
}
```

- And in `VirtualThreadsKafkaProducer class` will implement those using Virtual Thread and Apache-Kafka Library. For example, the same `send()` method may look similar to

```
Future<RecordMetadata> send(ProducerRecord<K,V> record) {
  Promise<RecordMetadata> promise = Promise.promise();
  eventQueue.add(new RP(record,promise)); //thread-safe queue
  if(!isRunning.get()) {  // isRunning is thread-safe AtomicBoolean
    isRunning.set(true);
    Thread.ofVirtual().start(() -> {
      while (queue.size() > 0) {
        try {
          RP rp = queue.take();  // RP consist promise with associated
          RecordMetadata metadata = producer.send(rp.record); // record
            rp.promise.complete(result);
          } catch (InterruptedException e) {
            rp.promise.fail(e);
          }
        }
```

```
            isRunning.set(false);
        });
    }
    return promise.future()
}
```

- Dispatcher

  For the Dispatcher, also the idea of the Consumer interface and class implementation is same as Producer. This ReactiveKafkaConsumer interface will also be implemented using both Vertx-Kafka-Client library and Virtual Thread with Apache-Kafka Library.

```
interface ReactiveKafkaConsumer {
    Future<ConsumerRecords<K,V>> poll(Duration timeout);
    Future<Void> close();
    Future<Void> pause(Collection<TopicPartition> partitions);
    Future<Void> resume(Collection<TopicPartition> partitions);
    // ... other consumer method signatures here
}
```

- Factory

  Both `KafkaProducerFactoy` and the `KafkaConsumerFactory` will return an object of `ReactiveKafkaProducer` and `ReactiveKafkaConsumer` respectively.

  And in implementation, we can decide on the parameter that we should create a VertxKafka Object or the VirtualThreadKafka object.

```
class KafkaProducerFactoyimpl implements KafkaProducerFactoy {
    @Override
    ReactiveKafkaProducer create(Vertx v, Properties props){
        return new VertxKafkaProducer(v,props);
    }
    @Override
    ReactiveKafkaProducer create(Properties props){
        return new VirtualThreadsKafkaProducer(props);
    }
}
```

4. Timeline

Milestones	Tasks	Date
1.	Before the official start date	
	1.1. Familiarise me more with the code. familiarize with the community and the processes. Learn more about Vert.x core, Solve other issues.	May 4 - May 28
2.	Migrate the Project to Java19	
	2.1. Upgrade the Java version to use JDK19 and related plugins.	May 29 - June 4
	2.2. Make sure everything will work fine, and fix any issue that arises because of JDK upgradation.	
3.	Implement our Reactive Minimal Interface using Vert.x Kafka client library	
	3.1. Identifies the Kafka functions and creates the Interface and the Factory files for both Producer and Consumer.	June 5 - June 11
	3.2. Create and implement VertxKafkaProducer class using the Vertx-Kafka-client library.	June 12 -June 21
	3.3. Create and implement VertxKafkaConsumer.	June 22 - July 02
	3.4. Write Unit tests for them.	July 03 - July 09
Midterm evaluation		July 10 - July 14
4.	Implement the interface using Virtual	

threads 4.1. Implement both the Producer and Consumer Interface using the native Apache-Kafka library with Virtual threads. 4.2. Write Unit tests for them.	July 14 - July 29 July 30 - Aug 06
5. Documentation 5.1. Write the Code implementation document. 5.2. Discussion of the results and implications of the performance comparison between both implementations.	Aug 07 - Aug 13
6. Time Buffer	Aug 14 - Aug 21
Submit & Final evaluation	Aug 21 - Aug 28

4. About Me

- Personal Information

Name: Debasish Biswas

Email: debasishbsws.abc@gmail.com

Contact: +91 6295887442

Location: India, Darjeeling.

TimeZone: GMT+5:30 (India Standard Time)

GitHub: github.com/debasishbsws

Twitter: twitter.com/debasishbsws

- Education

  University: Department of Computer Science, National Institute of Technology, Jamshedpur

- Obligation

  I will be able to work full-time (around 40 hours a week) from May till July end.
  I will have class exams in Aug, but I'm targeting a minimum commitment of 25 - 30 hrs a week.

Previous OpenSource Pull Requests

knative-sandbox/eventing-kafka-broker

- [Merged] Fix: Experemental filters AnyFilter working bug
- [Review] Fix: KafkaSink ContentMode not checking
- [Review] Allow retention.ms & cleanup.policy configuration of topic

knative/docs

- [Merged] Fix Code Block Formatting
- [Merged] Update NOT and CESQL which are not working

meshery/meshery

- [Merged] Improve the error messages in mesheryctl mesh

jenkinsci/authorize-project-plugin

- [Merged] Use https instead of git
