KIP-28 - Add a processor client

Status

Current state: *Adopted*

Discussion thread: here, here, here

JIRA: TBD

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

Motivation

A common use case for Kafka is real-time processes that transform data from input topics to output topics. Today there are a couple of options available for users to process such data:

1. Use the Kafka producer and consumer APIs with customized processing logic. For example:

```
// create a producer and a consumer
KafkaProducer producer = new KafkaProducer(configs);
KafkaConsumer consumer = new KafkaConsumer(configs);

// start a thread with a producer and consumer client
// for data IO and execute processing logic
new Thread(new Runnable {
  @Override
  void run() {
    while (isRunning) {
      // read some data from up-stream Kafka
      List<Message> inputMessages = consumer.poll();

      // do some processing..

      // send the output to the down-stream Kafka
      producer.send(outputMessages);
    }
  }
}).start()
```

2. Use a full-fledged stream processing system such as Storm, Samza, Spark Streaming, or Flink with Kafka as their source / sink stream data storage.

Both of those approaches have some downsides. Downsides of using the first option are that the producer and consumer APIs used for writing transformations are somewhat low level; simple examples are relatively simple, but any kind of more complex transformation tends to be a bit complicated. The opportunities for a richer client to add value beyond what the producer and consumer do would be:

1. Manage multi-threading and parallelism within a process.

2. Manage partitioning assignment to processes / threads.

3. Manage journaled local state storage.

4. Manage offset commits and "exactly-once" guarantees as appropriate features are added in Kafka to support this.

The second option, i.e. using a full stream processing framework can be a good solution but a couple of things tend to make it a bit heavy-weight (a brief and still-going survey can be found here):

1. These frameworks are poorly integrated with Kafka (different concepts, configuration, monitoring, terminology). For example, these frameworks only use Kafka as its stream data source / sink of the whole processing topology, while using their own in-memory format for storing intermediate data (RDD, Bolt memory map, etc). If users want to persist these intermediate results to Kafka as well, they need to break their processing into multiple topologies that need to be deployed separately, increasing operation and management costs.

2. These frameworks either duplicate or force the adoption of a packaging, deployment, and clustering solution. For example, in Storm you need to run a Storm cluster which is a separate thing that has to be monitored and operated. In an elastic environment like AWS, Mesos, YARN, etc this is sort of silly since then you have a Storm cluster inside the YARN cluster vs just directly running the jobs in Mesos; similarly Samza is tied up with YARN.

3. These frameworks can't be integrated with existing services or applications. For example, you can't just embed a light transformation library inside an existing app, but rather the entire framework that runs as a service.

Processor Client Proposal

We want to propose another standalone "processor" client besides the existing producer and consumer clients for processing data consumed from Kafka and storing results back to Kafka.

API Design

The processor client would provide the following APIs:

Data Processing

A processor computes on **a stream of messages**, with each message composed as a **key-value pair**.

Processor receives one message at a time and does not have access to the whole data set at once.

1. Per-message processing: this is the basic function that can be triggered once a new message has arrived from the stream.

2. Time-triggered processing: this function can be triggered whenever a specified time period has elapsed. It can be used for windowing computation, for example.

Compossible Processing

Multiple processors should be able to chained up to form a DAG (i.e. the processor **topology**) for complex processing logic.

Users can define such processor topology in a exploring REPL manner: make an initial topology, deploy and run, check the results and intermediate values, and pause the job and edit the topology on-the-fly.

Local State Storage

Users can create state storage inside a processor that can be accessed locally.

For example, a processor may retain a (usually most recent) subset of data for a join, aggregation / non-monolithic operations.

The proposed processing interface is as the following:

```
public interface ProcessorContext {

    void send(String topic, Object key, Object value);  // send the key value-pair to a Kafka topic

    void schedule(long timestamp);                      // repeatedly schedule the punctuation function for the
period

    void commit();                                      // commit the current state, along with the upstream
offset and the downstream sent data

    String topic();                                     // return the Kafka record's topic of the current
processing key-value pair

    int partition();                                    // return the Kafka record's partition id of the
current processing key-value pair

    long offset();                                      // return the Kafka record's offset of the current
processing key-value pair
}

public interface Processor<K, V>  {

    void init(ProcessorContext context);           // initialize the processor

    void process(K1 key, V1 value);                // process a key-value pair

    void punctuate();                              // process when the the scheduled time has reached

    void close();                                  // close the processor
}

public interface ProcessorDef {

    Processor instance();                          // create a new instance of the processor from its definition
}

public class TopologyBuilder {

    public final TopologyBuilder addSource(String name, String... topics) { ... }                         //
add a source node to the topology which generates incoming traffic with the specified Kafka topics

    public final TopologyBuilder addSink(String name, String topic, String... parentNames) { ... }        //
add a sink node to the topology with the specified parent nodes that sends out-going traffic to the specified
Kafka topics

    public final TopologyBuilder addProcessor(String name, ProcessorDef definition, String... parentNames) {
... }   // add a processor node to the topology with the specified parent nodes
}
```

And users can create their processing job with the created processor topology as the following:

```
public class ProcessorJob {

    private static class MyProcessorDef implements ProcessorDef {

        @Override
        public Processor<String, Integer> instance() {
            return new Processor<String, Integer>() {
                private ProcessorContext context;
                private KeyValueStore<String, Integer> kvStore;

                @Override
                public void init(ProcessorContext context) {
                    this.context = context;
                    this.context.schedule(this, 1000);
                    this.kvStore = new InMemoryKeyValueStore<>("local-state", context);
                }

                @Override
                public void process(String key, Integer value) {
                    Integer oldValue = this.kvStore.get(key);
                    if (oldValue == null) {
                        this.kvStore.put(key, value);
                    } else {
                        int newValue = oldValue + value;
                        this.kvStore.put(key, newValue);
                    }

                    context.commit();
                }

                @Override
                public void punctuate(long streamTime) {
                    KeyValueIterator<String, Integer> iter = this.kvStore.all();

                    while (iter.hasNext()) {
                        Entry<String, Integer> entry = iter.next();

                        System.out.println("[" + entry.key() + ", " + entry.value() + "]");

                        context.forward(entry.key(), entry.value());
                    }
                }

                @Override
                public void close() {
                    this.kvStore.close();
                }
            };
        }
    }

    public static void main(String[] args) throws Exception {
        StreamingConfig config = new StreamingConfig(new Properties());

        // build topology
        TopologyBuilder builder = new TopologyBuilder();
        builder.addSource("SOURCE", "topic-source");
               .addProcessor("PROCESS", new MyProcessorDef(), "SOURCE");
               .addSink("SINK", "topic-sink", "PROCESS");

        // start process
        KafkaStreaming streaming = new KafkaStreaming(builder, config);
        streaming.start();
    }
}
```

This example API demonstrates the abstraction of the low-level consumer / producer interfaces, such as consumer.poll() / commit(), producer.send (callback), producer.flush(), etc.

High-level Stream DSL

In addition to the processor API, we would also like to introduce a higher-level stream DSL for users that covers most common processor implementations.

```
public interface KStream<K, V> {

    /**
     * Creates a new stream consists of all elements of this stream which satisfy a predicate
     */
    KStream<K, V> filter(Predicate<K, V> predicate);

    /**
     * Creates a new stream by transforming key-value pairs by a mapper to all elements of this stream
     */
    <K1, V1> KStream<K1, V1> map(KeyValueMapper<K, V, K1, V1> mapper);

    /**
     * Creates a new stream by transforming valuesa by a mapper to all values of this stream
     */
    <V1> KStream<K, V1> mapValues(ValueMapper<V, V1> mapper);

    /**
     * Creates a new stream by applying a flat-mapper to all elements of this stream
     */
    <K1, V1> KStream<K1, V1> flatMap(KeyValueMapper<K, V, K1, ? extends Iterable<V1>> mapper);

    /**
     * Creates a new stream by applying a flat-mapper to all values of this stream
     */
    <V1> KStream<K, V1> flatMapValues(ValueMapper<V, ? extends Iterable<V1>> processor);

    /**
     * Creates a new windowed stream using a specified window instance.
     */
    KStreamWindowed<K, V> with(Window<K, V> window);

    /**
     * Creates an array of streams from this stream. Each stream in the array corresponds to a predicate in
     * supplied predicates in the same order.
     */
    KStream<K, V>[] branch(Predicate<K, V>... predicates);

    /**
     * Sends key-value to a topic.
     */
    void sendTo(String topic);

    /**
     * Sends key-value to a topic, also creates a new stream from the topic.
     * This is mostly used for repartitioning and is equivalent to calling sendTo(topic) and from(topic).
     */
    KStream<K, V> through(String topic);

    /**
     * Processes all elements in this stream by applying a processor.
     */
    <K1, V1> KStream<K1, V1> process(KafkaProcessor<K, V, K1, V1> processor);

    // .. more operators
}

public interface KStreamWindowed<K, V> extends KStream<K, V> {

    /**
     * Creates a new stream by joining this windowed stream with the other windowed stream.
```

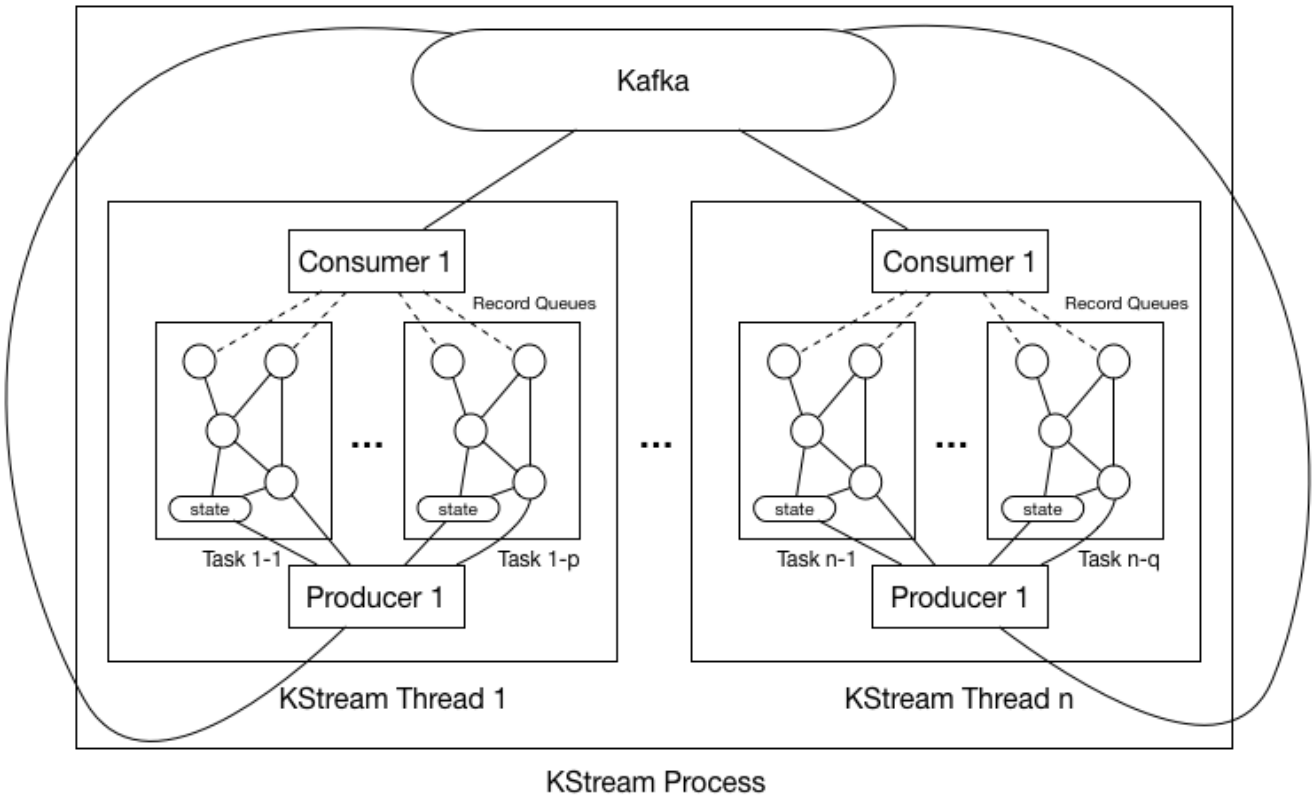
```
     * Each element arrived from either of the streams is joined with elements with the same key in another
stream.
     * The resulting values are computed by applying a joiner.
     */
    <V1, V2> KStream<K, V2> join(KStreamWindowed<K, V1> other, ValueJoiner<V, V1, V2> joiner);

    /**
     * Creates a new stream by joining this windowed stream with the other windowed stream.
     * Each element arrived from either of the streams is joined with elements with the same key in another
stream
     * if the element from the other stream has an older timestamp.
     * The resulting values are computed by applying a joiner.
     */
    <V1, V2> KStream<K, V2> joinPrior(KStreamWindowed<K, V1> other, ValueJoiner<V, V1, V2> joiner);
}
```

With this high-level interface, the user instantiated program can be simplified as (using lambda expression in Java 0.8):

```
public class KStreamJob {

    public static void main(String[] args) throws Exception {
        StreamingConfig config = new StreamingConfig(props);

        // build the topology
        KStreamBuilder builder = new KStreamBuilder();

        KStream<String, String> stream1 = builder.from("topic1");

        KStream<String, Integer> stream2 =
            stream1.map((key, value) -> new KeyValue<>(key, new Integer(value)))
                   .filter(((key, value) -> true));

        KStream<String, Integer>[] streams = stream2
            .branch((key, value) -> value > 10,
                    (key, value) -> value <= 10);

        streams[0].sendTo("topic2");
        streams[1].sendTo("topic3");

        // start the process
        KafkaStreaming kstream = new KafkaStreaming(builder, config);
        kstream.start();
    }
}
```

Architecture Design

We summarize some key architecture design points in the following sub-sections.



Partition Distribution

As shown in the digram above, each KStream process could have multiple threads (#.threads configurable in the properties), with each thread having a separate consumer and producer. So the first question is how can we distribute the partitions of the subscribed topics in the source processor among all the processes / threads.

There are a couple of common cases for partition management in KStream:

1. Co-partitioning: for windowed-joins.

2. Sticky partitioning: for stateful processing, users may want to have a static mapping from stream partitions to process threads.

3. N-way partitioning: when we have stand-by processor instances, users may want to assign a single stream partition to multiple process threads.

These use cases would require more flexible assignments than today's server-side strategies, so we need to extend the consumer coordinator protocol in the way that:

1. Consumers send JoinGroup with their subscribed topics, and receive the JoinGroup responses with **the list of members in the group** and **the list of topic-partitions**.

2. All consumers will get the same lists, and they can execute the same deterministic partition assignment algorithm to get their assigned topic-partitions.

With this new assignment protocol (details of this change can be found here), we distribute the partitions among worker thread as the following:

0. Upon starting the KStream process, user-specified number of KStream threads will be created. There is no shared variables between threads and no synchronization barriers as well hence these threads will execute completely asynchronously. Hence we will describe the behavior of a s single thread in all the following steps.

1. Thread constructs the user-specified processor topology without initializing it just in order to extract the list of **subscribed topics** from the topology.

2. Thread uses its consumer's partitionsFor() to fetch the metadata for each of the subscribed topics to get a information of topic -> #.partitions.

3. Thread now triggers consumer's subscribe() with the subscribed topics, which will then applies the new rebalance protocol. The join-group request will be instantiated as follows (for example):

```
JoinGroupRequest =>
  GroupId                 => "KStream-[JobName]"
  GroupType               => "KStream"
```

And the assignor interface is implemented as follows:

```
List<TopicPartition> assign(String consumerId,
                            Map<String, Integer> partitionsPerTopic,
                            List<ConsumerMetadata<T>> consumers) {

   // 1. trigger user-customizable grouping function to group the partitions into groups.
   // 2. assign partitions to consumers at the granularity of partition-groups.
   // 3*. persist the assignment result using commit-offset to Kafka.
}
```

The interface of the grouping function is the following, it is very similar to the assign() interface above, with the only difference that it does not have the consumer-lists.

```
interface PartitionGrouper {

  /**
   * Group partitions into partition groups
   */
  List<Set<TopicPartition>> group(Map<String, Integer> partitionsPerTopic)
}
```

So after the rebalance completes, each partition-group will be assigned as a whole to the consumers, i.e. no partitions belonging to the same group will be assigned to different consumers. The default grouping function maps partitions with the same id across topics to into a group (i.e. co-partitioning).

4. Upon getting the partition-groups, thread creates one **task** for each partition-group. And for each task, constructs the processor topology AND initializes the topology with the task context.

a. Initialization process will trigger Processor.init() on each processor in the topology following topology's DAG order.

b. All user-specified local states will also be created during the initialization process (we will talk about this later in the later sections).

c. Creates the record queue for each one of the task's associated partition-group's partitions, so that when consumers fetches new messages, it will put them into the corresponding queue.

Hence all tasks' topologies have the same "skeleton" but different processor / state instantiated objects; in addition, partitions are also synchronized at the tasks basis (we will talk about partition-group synchronization in the next section).

5. When rebalance is triggered, the consumers will read its last persisted partition assignment from Kafka and checks if the following are true when comparing with the new assignment result:

a. Existing partitions are still assigned to the same partition-groups.

b. New partitions are assigned to the existing partition-groups instead of creating new groups.

For a), since the partition-group's associated task-id is used as the corresponding change-log partition id, if a partition gets migrated from one group to another during the rebalance, its state will be no longer valid; for b) since we cannot (yet) dynamically change the #.partitions from the consumer APIs, dynamically adding more partition-groups (hence tasks) will cause change log partitions possibly not exist yet.

[NOTE] there are some more advanced partitioning setting such as sticky-partitioning / consistent hashing that we want to support in the future, which may then require additionally:

c. Partition groups are assigned to the specific consumers instead of randomly / round-robin.

Stream Time and Sync.

Time in the stream processing is very important. Windowing operations (join and aggregation) are defined by time. Since Kafka can replay stream, wall-clock based time (system time) may not make sense due to delayed messages / out-of-order messages. Hence we need to define a "time" for each stream according to its progress. We call it **stream time**.

Stream Time

A **stream** is defined to abstract all the partitions of the same topic within a task, and its name is the same as the topic name. For example if a task's assigned partitions are {Topic1-P1, Topic1-P2, Topic1-P3, Topic2-P1, Topic2-P2}, then we treat this task as having two streams: "Topic1" and "Topic2" where "Topic1" represents three partitions P1 P2 and P3 of Topic1, and stream "Topic2" represents two partitions P1 and P2 of Topic2.

Each message in a stream has to have a timestamp to perform window based operations and punctuations. Since Kafka message does not have timestamp in the message header, users can define a **timestamp extractor** based on message content that is used in the source processor when deserializing the messages. This extractor can be as simple as always returning the current system time (or wall-clock time), or it can be an Avro decoder that gets the timestamp field specified in the record schema.

In addition, since Kafka supports multiple producers sending message to the same topic, brokers may receive messages in order that is not strictly following their timestamps (i.e. out-of-order messages). Therefore, we cannot simply define the "stream time" as the timestamp of the currently processed message in the stream hence that time can move back and forth.

We define define the "stream time" as a monotonically increasing value as the following:

1. For each assigned partition, the thread maintains a record queue for buffering the fetched records from the consumer.

2. Each message has an associated timestamp that is extracted from the timestamp extractor in the message content.

3. The partition time is defined as **the lowest message timestamp value** in its buffer.

   a. When the lowest timestamp corresponding record gets processed by the thread, the partition time possibly gets advanced.

   b. The partition time will NOT gets reset to a lower value even if a later message was put in a buffer with a even lower timestamp.

4. The stream time is defined as the **lowest partition timestamp value across all its partitions in the task**:

   a. Since partition times are monotonically increasing, stream times are also monotonically increasing.

5. Any newly created streams through the upstream processors inherits the stream time of the parents; for joins, the bigger parent's stream time is taken.

Stream Synchronization

When joining two streams, their progress need to be synchronized. If they are out of sync, a time window based join becomes faulty. Say a delay of one stream is negligible and a delay of the other stream is one day, doing join over 10 minutes window does not make sense. To handle this case, we need to make sure that the consumption rates of all partitions within each task's assigned partition-group are "synchronized". Note that each thread may have one or more tasks, but it does not need to synchronize the partitions across tasks' partition-groups.

Work thread synchronizes the consumption within each one of such groups through consumer's pause / resume APIs as following:

1. When one un-paused partition is a head of time (partition time defined as above) beyond some defined threshold with other partitions, notify the corresponding consumer to pause.

2. When one paused partition is ahead of time below some defined with other partitions, notify the corresponding consumer to un-pause.

Two streams that are joined together have to be in the same task, and their represented partition lists have to match each other. That is, for example, a stream representing P1, P2 and P3 can be joined with another stream also representing P1, P2 and P3.

Local State Management

Users can create one or more state stores during their processing logic, and each task will have a **state manager** that keeps an instance of each specified store inside the task.

Since a single store instance will not be shared across multiple partition groups, and each partition group will only be processed by a single thread, this guarantees any store will not be accessed concurrently by multiple thread at any given time.

Log-backed State Storage

Each state store will be backed up by a different Kafka change log topic, and each instance of the store correlates to one partition of the topic, such that:

```
#. tasks == #. partition groups == #. store instances for each state store == #.partitions of the change log
for each state store
```

For example, if a processor instance consumes from upstream Kafka topic "topic-A" with 4 partitions, and creates two stores, namely store1 and store2, and user groups the 4 partitions into {topic-A-p1, topic-A-p2} and {topic-A-p3, topic-A-p4}; then two change log topics, for example namely "topic-store1-changelog" and "topic-store2-changelog", need to be created beforehand, each with two partitions.

After processor writes to a store instance, it first sends the change message to its corresponding changelog topic partition. When user calls commit() in his processor, KStream needs to flush both the store instance as well as the producer sending to the changelog, as well as committing the offset in the upstream Kafka. If these three operations cannot be done atomically, then if there is a crash in between this operations duplicates could be generated since the upstream Kafka committing offset is executed in the last step; if there three operations can be done atomically, then we can guarantee "exactly-once" semantics.

Persisting and Restoring State

When we close a KStream instance, the following steps are executed:

1. Flush all store's state as mentioned above.

2. Write the change log offsets for all stores into a **local offset checkpoint file**. The existence of the offset checkpoint file indicates if the instance was cleanly shutdown.

Upon (re-)starting the KStream instance:

1. Try to read the local offset checkpoint file into memory, and delete the file afterwards.

2. Check the offset of the corresponding change log partition read from the checkpoint file.

   a. If the offset is read successfully, load the previously flushed state and replay the change log from the read offset up to the log-end-offset.

   b. Otherwise, do not load the previously flushed state and replay the change log from the beginning up to the log-end-offset.

Workflow Summary

We summarize the workflow of a kafka streaming process following the above architecture design.

Startup

Upon user calling KafkaStreaming.start(), the process instance creates the worker threads given user specified #.threads. In each worker thread:

1. Construct the producer and consumer client, extract the subscription **topic names** from the topology.

2. Let the consumer to subscribe to the topics and gets the assigned partitions.

3. Trigger the **grouping** function with the assigned partitions get the returned list of **partition-groups (hence tasks) with associated ids**.

4. Initialize each task by:

   a. Creates a record queue for buffering the fetched records for each partition.

   b. Initialize a topology instance for the task from the builder with a newly created processor context.

   c. Initialize the state manager of the task and constructs / resumes user defined local states.

5. Runs the loop at its own pace until notified to be shutdown: there is no synchronization between these threads. In each iteration of the loop:

   a. Thread checks if the record queues are empty / low, and if yes calls consumer.poll(timeout) / consumer.poll(0) to re-fill the buffer.

   b. Choose one record from the queues and process it through the processor topology.

c. Check if some of the processors' punctuate functions need to be triggered, and if yes, execute the function.

d. Check if user calls commit() during the processing of this records; if yes commit the offset / flush the local state / flush the producer.

Shutdown

Upon user calling KafkaStreaming.shutdown(), the following steps are executed:

1. Commit / flush each partition-group's current processing state as described in the **local state management section.**

2. Close the embedded producer and consumer clients.

Packaging Design

It would be best to package Processor / KStream as a separate jar, since it introduces extra external dependencies, such as RocksDB, etc. Under this model:

1. We will let users to create their own MyKStream.java class that depends on the kafka-stream.jar.

2. We will let users to write their own Main function as the entry point for starting their process instance.

Current class / package names can be found in this PR. A general summary:

1. All classes are defined in the "streams" folder.

2. Low-level Processor interface is under the "o.a.k.streams.processor" package; high-level KStream interface is under the "o.a.k.streams.kstream" package.

3. Some example classes can be found in o.a.k.streams.examples.

4. Important internal classes include:

```
PartitionGroup: a set of partitions along with their queuing buffers and timestamp extraction logic.

ProcessorStateManager: the manager of the local states within a task.

ProcessorTopology: the instance of the topology generated by the TopologyBuilder.

StreamTask: the task of the processing tasks unit, which include a ProcessorTopology, a ProcessorStateManager
and a PartitionGroup.

StreamThread: contains multiple StreamTasks, a Consumer and a Producer client.

KStreamFilter/Map/Branch/...: implementations of high-level KStream topology builder operators.
```

Compatibility, Deprecation, and Migration Plan

This KIP only proposes additions. There should be no compatibility issues.