package com.twitter.unified\_user\_actions.service.module

import com.twitter.decider.Decider

import com.twitter.finagle.stats.Counter

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.finatra.kafka.producers.BlockingFinagleKafkaProducer

import com.twitter.finatra.kafka.serde.ScalaSerdes

import com.twitter.finatra.kafka.serde.UnKeyed

import com.twitter.finatra.kafka.serde.UnKeyedSerde

import com.twitter.kafka.client.headers.Implicits.\_

import com.twitter.kafka.client.headers.Zone

import com.twitter.kafka.client.processor.AtLeastOnceProcessor

import com.twitter.unified\_user\_actions.adapter.AbstractAdapter

import com.twitter.unified\_user\_actions.kafka.ClientConfigs

import com.twitter.unified\_user\_actions.kafka.ClientProviders

import com.twitter.unified\_user\_actions.thriftscala.UnifiedUserAction

import com.twitter.util.Duration

import com.twitter.util.Future

import com.twitter.util.StorageUnit

import com.twitter.util.logging.Logging

import org.apache.kafka.clients.consumer.ConsumerRecord

import org.apache.kafka.clients.producer.ProducerRecord

import org.apache.kafka.common.header.Headers

import org.apache.kafka.common.record.CompressionType

import org.apache.kafka.common.serialization.Deserializer

object KafkaProcessorProvider extends Logging {

lazy val actionTypeStatsCounterMap: collection.mutable.Map[String, Counter] =

collection.mutable.Map.empty

lazy val productSurfaceTypeStatsCounterMap: collection.mutable.Map[String, Counter] =

collection.mutable.Map.empty

def updateActionTypeCounters(

statsReceiver: StatsReceiver,

v: UnifiedUserAction,

topic: String

): Unit = {

val actionType = v.actionType.name

val actionTypeAndTopicKey = s"$actionType-$topic"

actionTypeStatsCounterMap.get(actionTypeAndTopicKey) match {

case Some(actionCounter) => actionCounter.incr()

case \_ =>

actionTypeStatsCounterMap(actionTypeAndTopicKey) =

statsReceiver.counter("uuaActionType", topic, actionType)

actionTypeStatsCounterMap(actionTypeAndTopicKey).incr()

}

}

def updateProductSurfaceTypeCounters(

statsReceiver: StatsReceiver,

v: UnifiedUserAction,

topic: String

): Unit = {

val productSurfaceType = v.productSurface.map(\_.name).getOrElse("null")

val productSurfaceTypeAndTopicKey = s"$productSurfaceType-$topic"

productSurfaceTypeStatsCounterMap.get(productSurfaceTypeAndTopicKey) match {

case Some(productSurfaceCounter) => productSurfaceCounter.incr()

case \_ =>

productSurfaceTypeStatsCounterMap(productSurfaceTypeAndTopicKey) =

statsReceiver.counter("uuaProductSurfaceType", topic, productSurfaceType)

productSurfaceTypeStatsCounterMap(productSurfaceTypeAndTopicKey).incr()

}

}

def updateProcessingTimeStats(statsReceiver: StatsReceiver, v: UnifiedUserAction): Unit = {

statsReceiver

.stat("uuaProcessingTimeDiff").add(

v.eventMetadata.receivedTimestampMs - v.eventMetadata.sourceTimestampMs)

}

def defaultProducer(

producer: BlockingFinagleKafkaProducer[UnKeyed, UnifiedUserAction],

k: UnKeyed,

v: UnifiedUserAction,

sinkTopic: String,

headers: Headers,

statsReceiver: StatsReceiver,

decider: Decider,

): Future[Unit] =

if (DefaultDeciderUtils.shouldPublish(decider = decider, uua = v, sinkTopic = sinkTopic)) {

updateActionTypeCounters(statsReceiver, v, sinkTopic)

updateProcessingTimeStats(statsReceiver, v)

// If we were to enable xDC replicator, then we can safely remove the Zone header since xDC

// replicator works in the following way:

// - If the message does not have a header, the replicator will assume it is local and

// set the header, copy the message

// - If the message has a header that is the local zone, the replicator will copy the message

// - If the message has a header for a different zone, the replicator will drop the message

producer

.send(

new ProducerRecord[UnKeyed, UnifiedUserAction](

sinkTopic,

null,

k,

v,

headers.remove(Zone.Key)))

.onSuccess { \_ => statsReceiver.counter("publishSuccess", sinkTopic).incr() }

.onFailure { e: Throwable =>

statsReceiver.counter("publishFailure", sinkTopic).incr()

error(s"Publish error to topic $sinkTopic: $e")

}.unit

} else Future.Unit

/\*\*

\* The default AtLeastOnceProcessor mainly for consuming from a single Kafka topic -> process/adapt -> publish to

\* the single sink Kafka topic.

\*

\* Important Note: Currently all sink topics share the same Kafka producer!!! If you need to create different

\* producers for different topics, you would need to create a customized function like this one.

\*/

def provideDefaultAtLeastOnceProcessor[K, V](

name: String,

kafkaSourceCluster: String,

kafkaGroupId: String,

kafkaSourceTopic: String,

sourceKeyDeserializer: Deserializer[K],

sourceValueDeserializer: Deserializer[V],

commitInterval: Duration = ClientConfigs.kafkaCommitIntervalDefault,

maxPollRecords: Int = ClientConfigs.consumerMaxPollRecordsDefault,

maxPollInterval: Duration = ClientConfigs.consumerMaxPollIntervalDefault,

sessionTimeout: Duration = ClientConfigs.consumerSessionTimeoutDefault,

fetchMax: StorageUnit = ClientConfigs.consumerFetchMaxDefault,

fetchMin: StorageUnit = ClientConfigs.consumerFetchMinDefault,

receiveBuffer: StorageUnit = ClientConfigs.consumerReceiveBufferSizeDefault,

processorMaxPendingRequests: Int,

processorWorkerThreads: Int,

adapter: AbstractAdapter[V, UnKeyed, UnifiedUserAction],

kafkaSinkTopics: Seq[String],

kafkaDestCluster: String,

kafkaProducerClientId: String,

batchSize: StorageUnit = ClientConfigs.producerBatchSizeDefault,

linger: Duration = ClientConfigs.producerLingerDefault,

bufferMem: StorageUnit = ClientConfigs.producerBufferMemDefault,

compressionType: CompressionType = ClientConfigs.compressionDefault.compressionType,

retries: Int = ClientConfigs.retriesDefault,

retryBackoff: Duration = ClientConfigs.retryBackoffDefault,

requestTimeout: Duration = ClientConfigs.producerRequestTimeoutDefault,

produceOpt: Option[

(BlockingFinagleKafkaProducer[UnKeyed, UnifiedUserAction], UnKeyed, UnifiedUserAction, String,

Headers, StatsReceiver, Decider) => Future[Unit]

] = None,

trustStoreLocationOpt: Option[String] = Some(ClientConfigs.trustStoreLocationDefault),

statsReceiver: StatsReceiver,

decider: Decider,

zone: Zone,

maybeProcess: (ConsumerRecord[K, V], Zone) => Boolean = ZoneFiltering.localDCFiltering[K, V] \_,

): AtLeastOnceProcessor[K, V] = {

lazy val singletonProducer = ClientProviders.mkProducer[UnKeyed, UnifiedUserAction](

bootstrapServer = kafkaDestCluster,

clientId = kafkaProducerClientId,

keySerde = UnKeyedSerde.serializer,

valueSerde = ScalaSerdes.Thrift[UnifiedUserAction].serializer,

idempotence = false,

batchSize = batchSize,

linger = linger,

bufferMem = bufferMem,

compressionType = compressionType,

retries = retries,

retryBackoff = retryBackoff,

requestTimeout = requestTimeout,

trustStoreLocationOpt = trustStoreLocationOpt,

)

mkAtLeastOnceProcessor[K, V, UnKeyed, UnifiedUserAction](

name = name,

kafkaSourceCluster = kafkaSourceCluster,

kafkaGroupId = kafkaGroupId,

kafkaSourceTopic = kafkaSourceTopic,

sourceKeyDeserializer = sourceKeyDeserializer,

sourceValueDeserializer = sourceValueDeserializer,

commitInterval = commitInterval,

maxPollRecords = maxPollRecords,

maxPollInterval = maxPollInterval,

sessionTimeout = sessionTimeout,

fetchMax = fetchMax,

fetchMin = fetchMin,

receiveBuffer = receiveBuffer,

processorMaxPendingRequests = processorMaxPendingRequests,

processorWorkerThreads = processorWorkerThreads,

adapter = adapter,

kafkaProducersAndSinkTopics =

kafkaSinkTopics.map(sinkTopic => (singletonProducer, sinkTopic)),

produce = produceOpt.getOrElse(defaultProducer),

trustStoreLocationOpt = trustStoreLocationOpt,

statsReceiver = statsReceiver,

decider = decider,

zone = zone,

maybeProcess = maybeProcess,

)

}

/\*\*

\* A common AtLeastOnceProcessor provider

\*/

def mkAtLeastOnceProcessor[K, V, OUTK, OUTV](

name: String,

kafkaSourceCluster: String,

kafkaGroupId: String,

kafkaSourceTopic: String,

sourceKeyDeserializer: Deserializer[K],

sourceValueDeserializer: Deserializer[V],

commitInterval: Duration = ClientConfigs.kafkaCommitIntervalDefault,

maxPollRecords: Int = ClientConfigs.consumerMaxPollRecordsDefault,

maxPollInterval: Duration = ClientConfigs.consumerMaxPollIntervalDefault,

sessionTimeout: Duration = ClientConfigs.consumerSessionTimeoutDefault,

fetchMax: StorageUnit = ClientConfigs.consumerFetchMaxDefault,

fetchMin: StorageUnit = ClientConfigs.consumerFetchMinDefault,

receiveBuffer: StorageUnit = ClientConfigs.consumerReceiveBufferSizeDefault,

processorMaxPendingRequests: Int,

processorWorkerThreads: Int,

adapter: AbstractAdapter[V, OUTK, OUTV],

kafkaProducersAndSinkTopics: Seq[(BlockingFinagleKafkaProducer[OUTK, OUTV], String)],

produce: (BlockingFinagleKafkaProducer[OUTK, OUTV], OUTK, OUTV, String, Headers, StatsReceiver,

Decider) => Future[Unit],

trustStoreLocationOpt: Option[String] = Some(ClientConfigs.trustStoreLocationDefault),

statsReceiver: StatsReceiver,

decider: Decider,

zone: Zone,

maybeProcess: (ConsumerRecord[K, V], Zone) => Boolean = ZoneFiltering.localDCFiltering[K, V] \_,

): AtLeastOnceProcessor[K, V] = {

val threadSafeKafkaClient =

ClientProviders.mkConsumer[K, V](

bootstrapServer = kafkaSourceCluster,

keySerde = sourceKeyDeserializer,

valueSerde = sourceValueDeserializer,

groupId = kafkaGroupId,

autoCommit = false,

maxPollRecords = maxPollRecords,

maxPollInterval = maxPollInterval,

sessionTimeout = sessionTimeout,

fetchMax = fetchMax,

fetchMin = fetchMin,

receiveBuffer = receiveBuffer,

trustStoreLocationOpt = trustStoreLocationOpt

)

def publish(

event: ConsumerRecord[K, V]

): Future[Unit] = {

statsReceiver.counter("consumedEvents").incr()

if (maybeProcess(event, zone))

Future

.collect(

adapter

.adaptOneToKeyedMany(event.value, statsReceiver)

.flatMap {

case (k, v) =>

kafkaProducersAndSinkTopics.map {

case (producer, sinkTopic) =>

produce(producer, k, v, sinkTopic, event.headers(), statsReceiver, decider)

}

}).unit

else

Future.Unit

}

AtLeastOnceProcessor[K, V](

name = name,

topic = kafkaSourceTopic,

consumer = threadSafeKafkaClient,

processor = publish,

maxPendingRequests = processorMaxPendingRequests,

workerThreads = processorWorkerThreads,

commitIntervalMs = commitInterval.inMilliseconds,

statsReceiver = statsReceiver

)

}

}