Avoid HTTP Request Duplicates with SCIO, a custom AsyncHttpParDoFn and State & Timers

Alberto López

BEAM SUMMIT

September 4-5, 2024

Sunnyvale, CA. USA

Agenda

- Introduction
- Problem statement
- Design
- Implementation: StateBaseAsyncDoFn
- Implementation: HttpClient
- Implementation: StateBaseAsyncDoFn & ZIORetry
- Testing
- Conclusions and Future Work

About me

- From Madrid, Spain.
- Lived in Ireland and England.
- Working in Deutsche Bank; Technology, Data and Innovation (TDI) as Technical Leader in Madrid.
- Electronics and Telecommunications Engineer.
  - Started coding in C, C++, Java and Android, +14 years ago.
  - Ended up doing loads of Scala the last 6 years (Kafka, Spark) and Beam (last 2 years).
- Music lover!

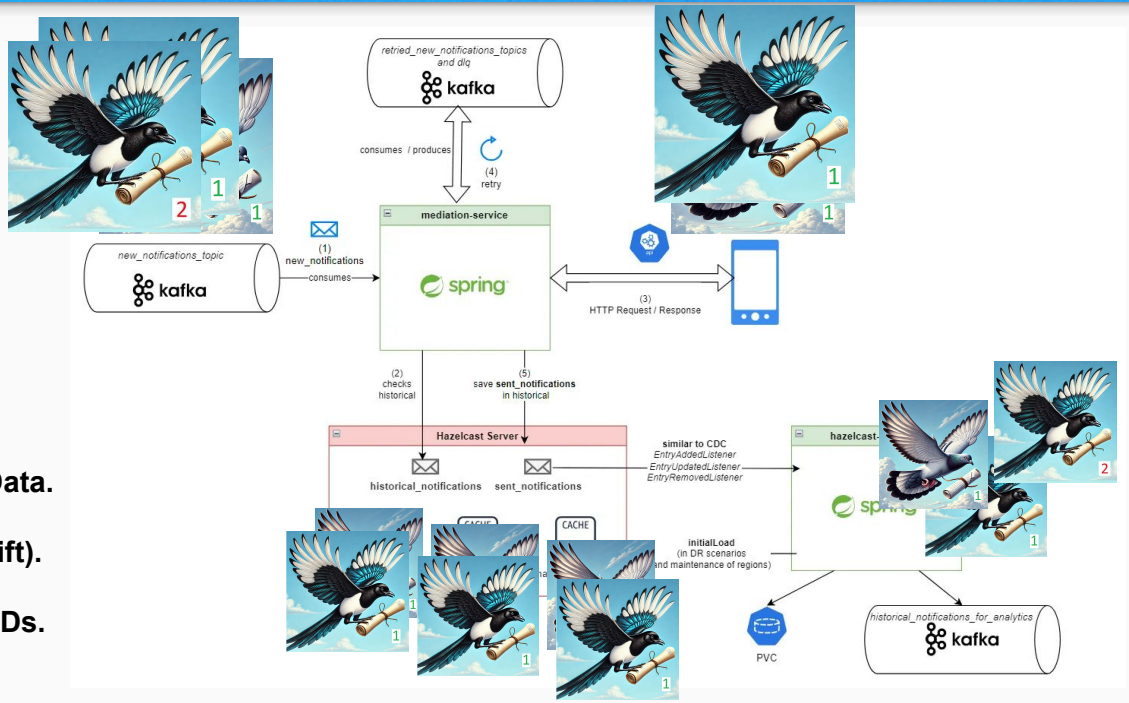
Introduction

BEAM SUMMIT

Introduction

We had to migrate an *on-premises microservice* (a.k.a **mediation-service**) application that:

1. **is implemented in Spring (living in Openshift).**
2. **reads: records from Kafka.**
3. **output: HTTP Requests to a notification hub endpoint.**
4. **keeps state (historical results) in a In Memory Data. Grid (IMDG) Hazelcast cluster (living in Openshift).**
5. **scales horizontally increasing its number of PODs.**

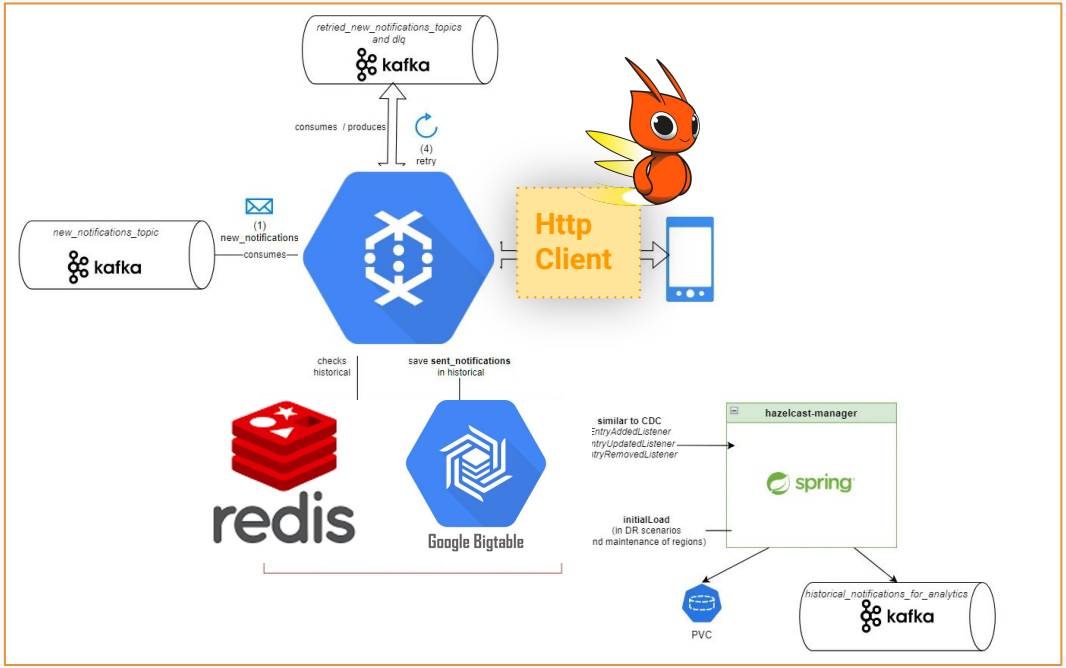


Problem Statement

BEAM SUMMIT

Problem Statement

1. **Could this design be simplified when migrating into GCP?**
2. **Could we use a distributed Big Data processing engine like Dataflow to undertake massive Async HTTP Requests?**
3. **What about keeping the state of this sent HTTP Requests to avoid duplicates?**

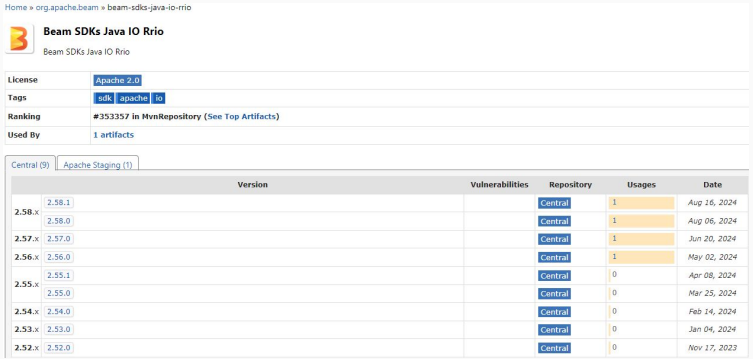


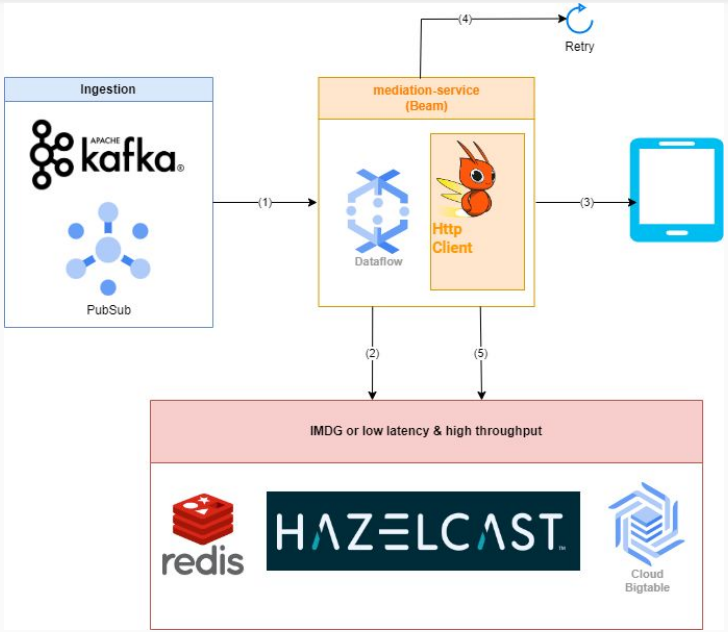
Design

BEAM SUMMIT

Design - Initial Assessment

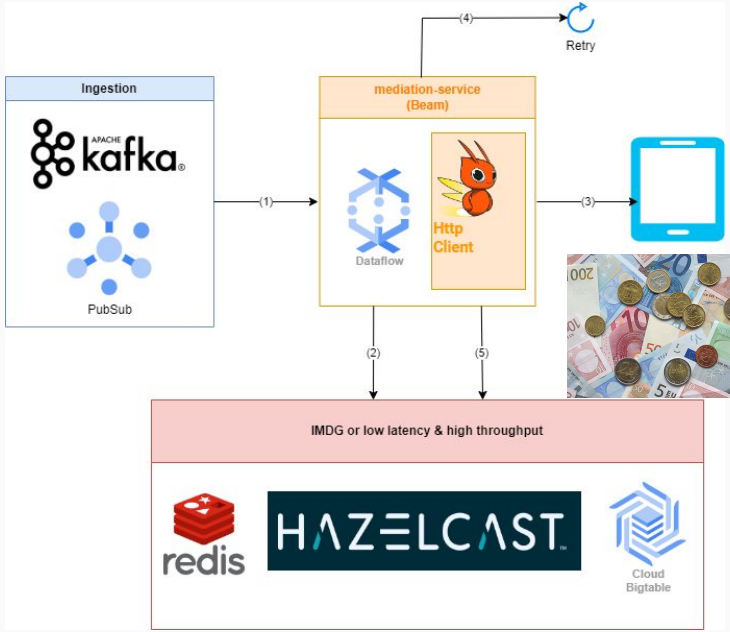
- State?
- Time To Live (Timer) ?
- Scalability?
- High throughput?
- Low latency lookups?
- Serverless?
- **HttpClient in Beam: HttpIO?** *(since beam 2.52.0)*
  https://beam.apache.org/documentation/io/built-in/webapis/

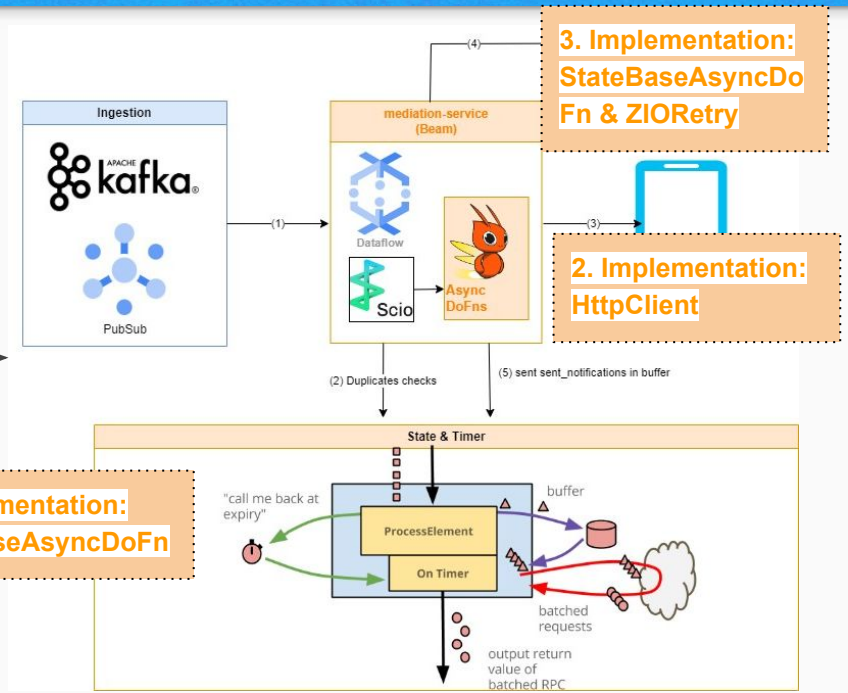




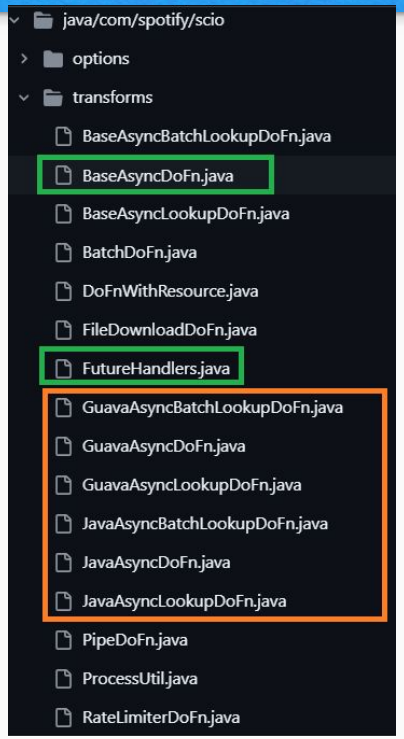
Avoid HTTP requests duplicates in Apache Beam with SCIO, a custom BaseAsyncDoFn and State and Timers

Design - Inception





Design - Why SCIO?



https://spotify.github.io/scio/

https://spotify.github.io/scio/releases/migrations/v0.8.0-Migration-Guide.html#async-dofns

https://cloud.google.com/blog/products/data-analytics/developing-beam-pipelines-using-scala/

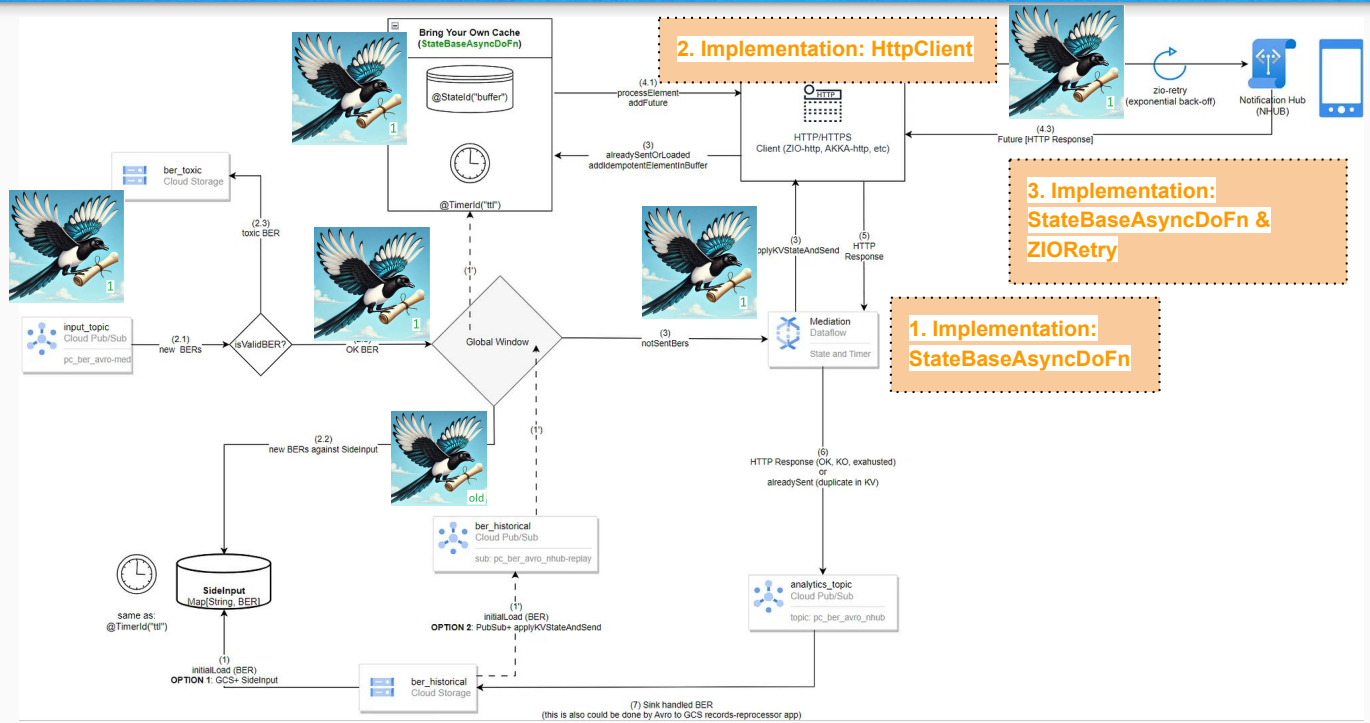
Scio

"Scio, pronounced shee-o, is Scala API for Beam developed by **Spotify** to build both Batch and Streaming pipelines."

Implementation

BEAM SUMMIT

Implementation



1. Reading **historical_notifications**
2. Ingestion and pre S & T Flow
3. applyKVState
4. Checks duplicates
   a.processElement
   b.StateAsyncParDoWithHttpHandler.scala
   -> HTTP request
   c.flush() in *StateBaseAsyncDoFn*.java:
   -> Future [Http Response]
5. Output HTTP Response: PubSub

Implementation: StateBaseAsyncDoFn

BaseAsyncDoFn.java (scio 0.14.6)

```
@ProcessElement
public void processElement(
    @Element InputT element,
    @Timestamp Instant timestamp,
    OutputReceiver<OutputT> out,
    BoundedWindow window) {
  flush(out);

  try {
    final UUID uuid = UUID.randomUUID();
    final FutureT future = processElement(element);
    futures.put(uuid, handleOutput(future, uuid, timestamp, window));
  } catch (Exception e) {
    LOG.error("Failed to process element", e);
    throw e;
  }
}
```

StateBaseAsyncDoFn.java

```
@ProcessElement
public void processElement(
        @Element InputT element,
        @TimerId("ttl") Timer ttl,
        @StateId("buffer") BagState<InputT> buffer,
        @Timestamp Instant timestamp,
        OutputReceiver<OutputT> out,
        BoundedWindow window) {
    flush(out);
    settingElementTTLTimer(buffer, ttl, element); // abstract
    initialLoad(buffer, element, ttl); // abstract

    if (!alreadySentOrLoaded(buffer, element, ttl)) // abstract
        try {
            final UUID uuid = UUID.randomUUID();
            addIdempotentElementInBuffer(buffer, element);// abstract, WATCH OUT: potential race conditions
            final FutureT future = processElement(element);
            // STEP 4.1)
            futures.put(uuid, handleOutput(future, element, buffer, uuid, timestamp, window));
        } catch (Exception e) {
            LOG.error("Failed to process element", e);
            throw e;
        }
    else
        outputAlreadySentOrLoaded(element, out); // abstract
}

protected abstract void outputAlreadySentOrLoaded(InputT element, OutputReceiver<OutputT> out);

protected abstract void initialLoad(@StateId("buffer") BagState<InputT> buffer, InputT element, @TimerId("ttl") Timer ttl);

protected abstract boolean alreadySentOrLoaded(@StateId("buffer") BagState<InputT> buffer, InputT element, @TimerId("ttl") Timer ttl);

protected abstract void settingElementTTLTimer(@StateId("buffer") BagState<InputT> buffer, @TimerId("ttl") Timer ttl, InputT element);
```

StateBaseAsyncDoFn.java:

- is mainly based on the implementation of SCIO's BaseAsyncDoFn.java: https://spotify.github.io/scio/releases/migrations/v0.8.0-Migration-Guide.html#async-dofns .
- abstracts out some new methods.

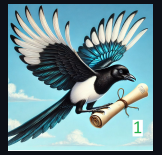
Implementation: HttpClient

```scala
implicit val lastDeviceReqEncoder: JsonEncoder[MyHttpRequest.HttpRequest] = DeriveJsonEncoder.gen[MyHttpRequest.HttpRequest]
implicit val responseDecoder: JsonDecoder[NotificationResponse] = DeriveJsonDecoder.gen[NotificationResponse]
// https://zio.dev/zio-http/examples/basic/https-client
val httpsLayers = ZClient.default ++
  NettyClientDriver.live ++
  DnsResolver.default ++
  ZLayer.succeed(NettyConfig.default) ++
  Scope.default
val httpLayers = ZClient.default ++
  Scope.default

def sendPushWithRequest(record: InputBer) = for {
  requestDto <- ZIO.succeed(getRequest(record))
  responseDto <- sendPushWithResponse(requestDto, record)
} yield responseDto

def sendPushWithResponse(
                          requestDto: MyHttpRequest.HttpRequest,
                          record: InputBer
                        ) = {
  val requestJson = requestDto.toJson

  val zioRequest = Request(
    url = url,
    method = Method.POST,
    headers = Headers("Content-Type" -> "application/json"),
    body = Body.fromString(requestJson)
  )

  val beforeNhubTs = jodaNowGetMillis
  log.info(s"zioRequest=$zioRequest")
  for {
    response <- {
      ZClient
        .request(zioRequest)
        .provideLayer(httpLayers)
    }
    responseBody <- response.body.asString
    responseDto <- ZIO
      .fromEither(responseBody.fromJson[NotificationResponse])
      .mapError(new RuntimeException(_)) // Converting parsing error to Throwable
  } yield KV.of(newEventRecordOK(record, beforeNhubTs, responseDto.body), responseDto)
}
```

ZIO (WIP)

```scala
object AkkaHttpClient extends Serializable {

  import akka.http.scaladsl.settings.ConnectionPoolSettings
  import com.db.myproject.mediation.MediationService.{akkaConfig, domain, mediationConfig, fullUrl, url}
  import scala.concurrent.duration._

  lazy val connectionPoolSettings = ConnectionPoolSettings(system)
    .withMaxOpenRequests(akkaConfig.maxOpenRequests)
    .withMaxConnections(akkaConfig.maxOpenConnection)

  private val log: Logger = LoggerFactory.getLogger(getClass.getName)
  implicit lazy val sslConfig = MySslConfig(mediationConfig.mediation.sslConfigPath.get, mediationConfig.gcp.project).sslConfig
  lazy val httpsSSLConnectionContext = AkkaSSLContextFromSecretManager.httpsConnectionContext

  lazy val poolClientFlow = poolClient
    .initialTimeout(FiniteDuration(akkaConfig.initialTimeout, TimeUnit.SECONDS))
    .completionTimeout(FiniteDuration(akkaConfig.completionTimeout, TimeUnit.SECONDS))
    .buffer(akkaConfig.buffer, OverflowStrategy.backpressure)
    .throttle(akkaConfig.throttleRequests, akkaConfig.throttlePerSecond.second, akkaConfig.throttleBurst, ThrottleMode.Shaping)

  lazy val poolClient: Flow[(HttpRequest, Any), (Try[HttpResponse], Any), Any] =
    if (fullUrl.contains("https")) {
      mediationConfig.mediation.endpoint.certEnabled match {
        case true => Http().cachedHostConnectionPoolHttps(url, connectionContext = httpsSSLConnectionContext, settings = connectionPoolSettings)
        case false => Http().cachedHostConnectionPoolHttps(url, settings = connectionPoolSettings)
      }
    } else Http().cachedHostConnectionPool(url, settings = connectionPoolSettings)

  implicit val system: ActorSystem = ActorSystem()
  implicit val materializer: ActorMaterializer = ActorMaterializer()
  implicit val ec: ExecutionContext = system.dispatcher
```

AKKA



Avoid HTTP requests duplicates in Apache Beam with SCIO, a custom BaseAsyncDoFn and State and Timers

BEAM SUMMIT

Implementation: AkkaHttpClient

```
def sendPush(record: InputBer): Future[(Try[HttpResponse], Any)] =
  getResponseFuture(
    getRequest(record)
      .asInstanceOf[MyHttpRequest.HttpRequest]
      .toJson
      .toString,
    absoluteURL(mediationConfig)
  )

def getResponseFuture(reqRawString: String, url: String): Future[(Try[HttpResponse], Any)] = {
  log.info(s"reqRawString=$reqRawString")
  val entity = HttpEntity(MediaTypes.`application/json`, ByteString(reqRawString))
  val httpRequest = HttpRequest(
    method = HttpMethods.POST,
    uri = url,
    entity = entity
  )
  log.debug(s"httpRequest=${httpRequest}")
  Source
    .single(httpRequest -> ())
    .via(AkkaHttpClient.poolClientFlow)
    .runWith(Sink.head)
}
```

GIT:
scio-db/src/main/scala/com.db.myproject/mediation/http/StateAsyncParDoWithHttpHandler.scala at cleanup-for-medium · albertols/scio-db (github.com)

sendPushWithFutureResponse(ber) , how ???

Wrap it as Future!

```
class AkkaHttpClient extends AbstractHttpClient {

  import AkkaHttpClient._
  override def sendPushWithFutureResponse(ber: OutputBer): FutureKVOutputBerAndHttpResponse = {
    import akka.http.scaladsl.marshallers.sprayjson.SprayJsonSupport._
    val beforeNhubTs = jodaNowGetMillis
    sendPush(ber)
      .flatMap {
        case (Success(response), _) =>
          log.info(s"Unmarshalling $response")
          Unmarshal(response.entity).to[NotificationResponse]
        case (Failure(ex), _) =>
          Future.failed(new Exception(s"NHUB Request failed with exception: $ex"))
      }
      .map(k => KV.of(newBerWithLastNHubTimestamp(ber, beforeNhubTs), k))
  }
}
```

```
// needed due to StateBaseAsyncDoFn
type FutureKVOutputBerAndHttpResponse = Future[KVOutputBerAndHttpResponse]
```

Implementation: StateBaseAsyncDoFn+ZIORetry

```
lazy val berKVstate = new StateAsyncParDoWithHttpHandler(mediationConfig, true)
def applyBerKVstate(
  berKVstate: StateAsyncParDoWithHttpHandler,
  windowedBers: SCollection[KV[String, MyEventRecord]]
): SCollection[StateAndTimerType.KVOutputBerAndHttpResponse] = windowedBers.applyTransform(ParDo.of(berKVstate))
```

applying S & T

```
class StateAsyncParDoWithHttpHandler(mediationConfig: MediationConfig, applyInitialLoad: Boolean)
    extends StateScalaAsyncDoFn[
      StateAndTimerType.KVInputStringAndBer,
      StateAndTimerType.KVOutputBerAndHttpResponse,
      AbstractHttpClient
    ] {
  val log: Logger = LoggerFactory getLogger getClass.getName
  val MAX_RETRIES = 3
  val BACKOFF_SECONDS = 10
  private val BUFFER_TIME = Duration.standardSeconds(mediationConfig.mediation.ttlTime)
  implicit lazy val zioRuntime = Runtime.default

  lazy val httpClient = {
    mediationConfig.mediation.httpClientType match {
      case "akka" => new AkkaHttpClient
      case "zio"  => new ZioHttpClient
    }
  }

  override def createResource(): AbstractHttpClient = httpClient

  override def getResourceType: ResourceType = ResourceType.PER_CLASS

  override def processElement(
    input: StateAndTimerType.KVInputStringAndBer
  ): StateAndTimerType.FutureKVOutputBerAndHttpResponse =
    // STEP 4.2)
    Try(mediationConfig.mediation.retryNotifications match {
      case true  => sendPushWithRetryZio(input.getValue)
      case false => getResource.sendPushWithFutureResponse(input.getValue)
    }) match {
      case Success(s) => s
      case Failure(ex) =>
        import scala.concurrent.ExecutionContext.Implicits.global
        Future(KV.of(input.getValue, koNotificationResponse(s"NOTIFICATION_ERROR: ${ex}")))
    }
```

uses

```
import com.spotify.scio.transforms.ScalaFutureHandlers

import scala.concurrent.Future

/**
 * A [[org.apache.beam.sdk.transforms.DoFn DoFn]] that handles asynchronous requests to an external service that returns
 * Scala [[Future]] s.
 */
abstract class StateScalaAsyncDoFn[I, O, R]
    extends StateBaseAsyncDoFn[I, O, R, Future[O]]
    with ScalaFutureHandlers[O] {}
```

sendPushWithFutureResponse(ber) , where ???

But retries… how ???

```
def sendPushWithRetryZio(
  record: InputBer
)(implicit zioRuntime: Runtime[Any]): StateAndTimerType.FutureKVOutputBerAndHttpResponse = {
  import zio._
  lazy val futureRetriableBer = ZIO
    .attempt {
      getResource.sendPushWithFutureResponse(newEventRecordWithRetryIncrement(record))
    }
    .retry(Schedule.fixed(BACKOFF_SECONDS.second) && Schedule.recurs(MAX_RETRIES))
    .onError(cause =>
      ZIO.succeed(
        log.error(s"[exhausted_notification=${record.getEvent.getTransactionId}] Retried error:${cause}", cause)
      )
    )

  Unsafe.unsafe { implicit unsafe =>
    zioRuntime.unsafe.run(futureRetriableBer).getOrThrowFiberFailure()
  }
}
```

Testing

BEAM SUMMIT

Unit Testing

```
"1 OK and 2 SENT_OR_DUPLICATED HTTP_RESPONSE" should "exist in the same stream" in {
  // simulates similar stream as PubSub
  val streamWithDuplicates: TestStream[MyEventRecord] = testStreamOf[MyEventRecord]
    // Start at the epoch
    .advanceWatermarkTo(baseTime)
    // add some elements ahead of the watermark
    .addElements(event(not_sent_debit_quique, Duration.standardSeconds(1)))
    // advance the watermark
    .advanceWatermarkTo(baseTime.plus(Duration.standardSeconds(10)))
    // x 2 duplicated elements
    .addElements(event(not_sent_debit_quique, Duration.standardSeconds(5)))
    .addElements(event(not_sent_debit_quique, Duration.standardSeconds(1)))
    .advanceWatermarkToInfinity
```

scio-db/src/test/scala/mediation/MediationServiceSpec.scala at cleanup-for-medium · albertols/scio-db (github.com)

```
runWithContext { sc =>
  val okNotSentBers: SCollection[KV[String, MyEventRecord]] = MediationService
    .mapWithIdempotentKeyAndGlobalWindow(
      sc.testStream(streamWithDuplicates),
      ber => getIdempotentNotificationKey(ber)
    )
    // .distinctByKey // not applied as it would get rid of the same records in GlobalWindow
    .map { ber =>
      KV.of(ber._1, ber._2)
    }
  val appliedState: SCollection[StateAndTimerType.KVOutputBerAndHttpResponse] = applyBerKVState(berKVState, okNotSentBers)
  val bersForAnalytics: SCollection[(MyHttpResponse.NotificationResponse, MyEventRecord)] = bersAfterHttpResponse(appliedState)
  val httpResponses: SCollection[MyHttpResponse.NotificationResponse] = bersForAnalytics.keys

  httpResponses should {
    val expectedOkHttpResponse: MyHttpResponse.NotificationResponse = MyHttpResponse.NotificationResponse(
      101, // id automatically returned by jsonplaceholder.typicode.com
      title = not_sent_debit_quique.getNotification.getId.toString,
      body = not_sent_debit_quique.getNotification.getMessage.toString,
      userId = not_sent_debit_quique.getCustomer.getId.toString.toInt
    )
    containInAnyOrder(Seq(expectedOkHttpResponse, SENT_OR_DUPLICATED, SENT_OR_DUPLICATED))
```

Stress Testing

```
akka {
max-open-requests = 20000
max-open-connection = 20000
initial-timeout = 30
completion-timeout = 60
buffer = 20000
throttle-requests = 1000
throttle-per-second = 1
throttle-burst = 1000
}
```

+200 K per min peaks

- e2-highmem-4: scaling was needed (maxWorkers=3).

- e2-standard-4: we got good balance between scalability, latency and cost. Scaling was not needed.

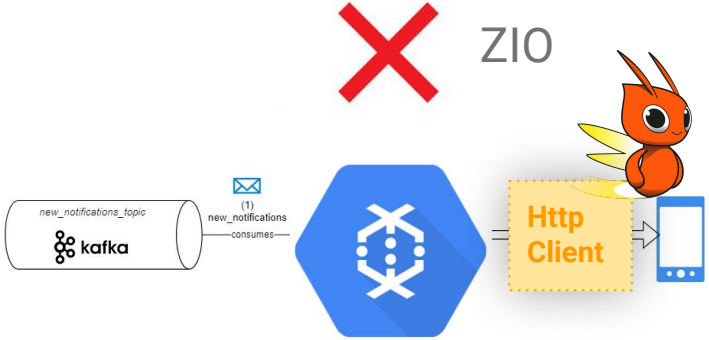
- e2-highcpu-4: we got some OutOfMemory issues (restarting a worker), it did handle the load but with the worst performance.

Conclusions and Future Work

BEAM SUMMIT

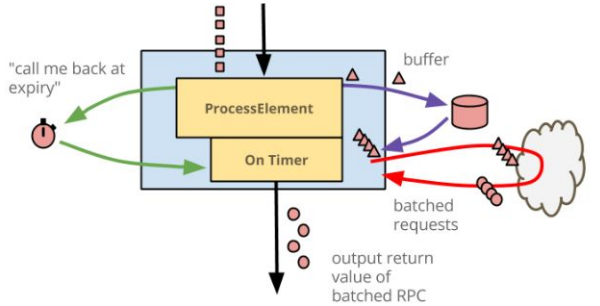
Conclusions

Cost saving



Performance

- High throughput for notification processing and low latency for reading/writing notifications by applying the State & Timer pattern.
- Great horizontal scalability.
- High availability and robustness.



Future Work

- Loading historical_notifications within the State & Timer (**PubSub**)

- Brining **more HTTP clients** options and comparing performance among them.

- Future similar features in Beam or SCIO core? (since beam 2.52.0):

    https://beam.apache.org/documentation/io/built-in/**webapis**/

    https://github.com/spotify/scio/issues/5055

Do not miss out tomorrow !
Drools ParDo and SCIO Dataflow: A Goodbye Microservices Tale

Kudos

- Intro to State and Timer (from Beam Summit 2019):
https://www.youtube.com/watch?v=Q_v5Zsjuuzg

- Great intro to SCIO Beam (from Beam Summit 2021):
https://github.com/iht/scio-scala-beam-summit

- State and Timer (from Beam Summit 2023)

https://beamsummit.org/sessions/2023/too-big-to-fail-a-beam-pattern-for-enriching-a-stream-using-state-and-timers/

- Unbreakable & Supercharged Beam Apps with Scala + ZIO (from Beam Summit 2023)
https://beamsummit.org/sessions/2023/unbreakable-supercharged-beam-apps-with-scala-zio/

By these Beam Summit contributors:

Kenneth Knowles:
https://beamsummit.org/speakers/kenneth-knowles/

Reza Rokni:
https://2021.beamsummit.org/speakers/reza-rokni/

Israel Herraiz:
https://github.com/iht

Tobias Kaymak:
https://beamsummit.org/speakers/tobias-kaymak/

Aris Vlasakakis:
https://beamsummit.org/speakers/aris-vlasakakis/

Sahil Khandwala:
https://beamsummit.org/speakers/sahil-khandwala/

Thank you!

Questions?

Medium Post:
https://medium.com/@serna.alberto.eng/avoid-http-requests-duplicates-in-apache-beam-with-scio-a-custom-baseasyncdofn-and-state-and-2c7d63059ab3

LinkedIn:
https://www.linkedin.com/in/albertolose