package com.twitter.recos.user\_tweet\_graph

import com.twitter.abdecider.ABDeciderFactory

import com.twitter.abdecider.LoggingABDecider

import com.twitter.app.Flag

import com.twitter.conversions.DurationOps.\_

import com.twitter.finagle.ThriftMux

import com.twitter.finagle.http.HttpMuxer

import com.twitter.finagle.mtls.authentication.ServiceIdentifier

import com.twitter.finagle.mtls.client.MtlsStackClient.MtlsThriftMuxClientSyntax

import com.twitter.finagle.mtls.server.MtlsStackServer.\_

import com.twitter.finagle.mux.ClientDiscardedRequestException

import com.twitter.finagle.mux.transport.OpportunisticTls

import com.twitter.finagle.service.ReqRep

import com.twitter.finagle.service.ResponseClass

import com.twitter.finagle.thrift.ClientId

import com.twitter.finatra.kafka.consumers.FinagleKafkaConsumerBuilder

import com.twitter.finatra.kafka.domain.KafkaGroupId

import com.twitter.finatra.kafka.domain.SeekStrategy

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

import com.twitter.frigate.common.util.ElfOwlFilter

import com.twitter.frigate.common.util.ElfOwlFilter.ByLdapGroup

import com.twitter.graphjet.bipartite.MultiSegmentPowerLawBipartiteGraph

import com.twitter.logging.\_

import com.twitter.recos.decider.EndpointLoadShedder

import com.twitter.recos.decider.UserTweetGraphDecider

import com.twitter.recos.graph\_common.FinagleStatsReceiverWrapper

import com.twitter.recos.graph\_common.MultiSegmentPowerLawBipartiteGraphBuilder

import com.twitter.recos.internal.thriftscala.RecosHoseMessage

import com.twitter.recos.user\_tweet\_graph.RecosConfig.\_

import com.twitter.recos.user\_tweet\_graph.relatedTweetHandlers.ConsumersBasedRelatedTweetsHandler

import com.twitter.recos.user\_tweet\_graph.relatedTweetHandlers.ProducerBasedRelatedTweetsHandler

import com.twitter.recos.user\_tweet\_graph.relatedTweetHandlers.TweetBasedRelatedTweetsHandler

import com.twitter.recos.user\_tweet\_graph.store.UserRecentFollowersStore

import com.twitter.server.Deciderable

import com.twitter.server.TwitterServer

import com.twitter.server.logging.{Logging => JDK14Logging}

import com.twitter.servo.request.\_

import com.twitter.servo.util.ExceptionCounter

import com.twitter.simclusters\_v2.common.UserId

import com.twitter.socialgraph.thriftscala.SocialGraphService

import com.twitter.storehaus.ReadableStore

import com.twitter.util.Await

import com.twitter.util.Duration

import com.twitter.util.JavaTimer

import com.twitter.util.Throw

import com.twitter.util.Timer

import java.net.InetSocketAddress

import java.util.concurrent.TimeUnit

import org.apache.kafka.clients.CommonClientConfigs

import org.apache.kafka.common.config.SaslConfigs

import org.apache.kafka.common.config.SslConfigs

import org.apache.kafka.common.security.auth.SecurityProtocol

import org.apache.kafka.common.serialization.StringDeserializer

import scala.reflect.ClassTag

object Main extends TwitterServer with JDK14Logging with Deciderable {

profile =>

val shardId: Flag[Int] = flag("shardId", 0, "Shard ID")

val servicePort: Flag[InetSocketAddress] =

flag("service.port", new InetSocketAddress(10143), "Thrift service port")

val logDir: Flag[String] = flag("logdir", "recos", "Logging directory")

val numShards: Flag[Int] = flag("numShards", 1, "Number of shards for this service")

val truststoreLocation: Flag[String] =

flag[String]("truststore\_location", "", "Truststore file location")

val hoseName: Flag[String] =

flag("hosename", "recos\_injector\_user\_user", "the kafka stream used for incoming edges")

val dataCenter: Flag[String] = flag("service.cluster", "atla", "Data Center")

val serviceRole: Flag[String] = flag("service.role", "Service Role")

val serviceEnv: Flag[String] = flag("service.env", "Service Env")

val serviceName: Flag[String] = flag("service.name", "Service Name")

private val maxNumSegments =

flag("maxNumSegments", graphBuilderConfig.maxNumSegments, "the number of segments in the graph")

private val statsReceiverWrapper = FinagleStatsReceiverWrapper(statsReceiver)

/\*\*

\* A ClientRequestAuthorizer to be used in a request-authorization RequestFilter.

\*/

lazy val clientAuthorizer: ClientRequestAuthorizer =

ClientRequestAuthorizer.observed(

ClientRequestAuthorizer.permissive,

new ClientRequestObserver(statsReceiver)

)

lazy val clientId = ClientId(s"usertweetgraph.${serviceEnv()}")

private def makeThriftClient[ThriftServiceType: ClassTag](

dest: String,

label: String,

serviceIdentifier: ServiceIdentifier,

requestTimeout: Duration = 100.milliseconds

): ThriftServiceType = {

ThriftMux.client

.withClientId(ClientId("usertweetgraph.prod"))

.withOpportunisticTls(OpportunisticTls.Required)

.withMutualTls(serviceIdentifier)

.withRequestTimeout(requestTimeout)

.withStatsReceiver(statsReceiver.scope("clnt"))

.withResponseClassifier {

case ReqRep(\_, Throw(\_: ClientDiscardedRequestException)) => ResponseClass.Ignorable

}.build[ThriftServiceType](dest, label)

}

private val shutdownTimeout = flag(

"service.shutdownTimeout",

5.seconds,

"Maximum amount of time to wait for pending requests to complete on shutdown"

)

/\*\*

\* ExceptionCounter for tracking failures from RequestHandler(s).

\*/

lazy val exceptionCounter = new ExceptionCounter(statsReceiver)

/\*\*

\* Function for translating exceptions returned by a RequestHandler. Useful

\* for cases where underlying exception types should be wrapped in those

\* defined in the project's Thrift IDL.

\*/

lazy val translateExceptions: PartialFunction[Throwable, Throwable] = {

case t => t

}

// \*\*\*\*\*\*\*\*\* logging \*\*\*\*\*\*\*\*\*\*

lazy val loggingLevel: Level = Level.INFO

lazy val recosLogPath: String = logDir() + "/recos.log"

lazy val graphLogPath: String = logDir() + "/graph.log"

lazy val accessLogPath: String = logDir() + "/access.log"

override def loggerFactories: List[LoggerFactory] =

List(

LoggerFactory(

level = Some(loggingLevel),

handlers = QueueingHandler(

handler = FileHandler(

filename = recosLogPath,

level = Some(loggingLevel),

rollPolicy = Policy.Hourly,

rotateCount = 6,

formatter = new Formatter

)

) :: Nil

),

LoggerFactory(

node = "graph",

useParents = false,

level = Some(loggingLevel),

handlers = QueueingHandler(

handler = FileHandler(

filename = graphLogPath,

level = Some(loggingLevel),

rollPolicy = Policy.Hourly,

rotateCount = 6,

formatter = new Formatter

)

) :: Nil

),

LoggerFactory(

node = "access",

useParents = false,

level = Some(loggingLevel),

handlers = QueueingHandler(

handler = FileHandler(

filename = accessLogPath,

level = Some(loggingLevel),

rollPolicy = Policy.Hourly,

rotateCount = 6,

formatter = new Formatter

)

) :: Nil

),

LoggerFactory(

node = "client\_event",

level = Some(loggingLevel),

useParents = false,

handlers = QueueingHandler(

maxQueueSize = 10000,

handler = ScribeHandler(

category = "client\_event",

formatter = BareFormatter

)

) :: Nil

)

)

// \*\*\*\*\*\*\*\* Decider \*\*\*\*\*\*\*\*\*\*\*\*\*

// \*\*\*\*\*\*\*\*\* ABdecider \*\*\*\*\*\*\*\*\*\*

val abDeciderYmlPath: String = "/usr/local/config/abdecider/abdecider.yml"

val scribeLogger: Option[Logger] = Some(Logger.get("client\_event"))

val abDecider: LoggingABDecider =

ABDeciderFactory(

abDeciderYmlPath = abDeciderYmlPath,

scribeLogger = scribeLogger,

environment = Some("production")

).buildWithLogging()

// \*\*\*\*\*\*\*\*\* Recos service \*\*\*\*\*\*\*\*\*\*

def main(): Unit = {

log.info("building graph with maxNumSegments = " + profile.maxNumSegments())

implicit val timer: Timer = new JavaTimer(true)

val graph = MultiSegmentPowerLawBipartiteGraphBuilder(

graphBuilderConfig.copy(maxNumSegments = profile.maxNumSegments()),

statsReceiverWrapper

)

val kafkaConfigBuilder = FinagleKafkaConsumerBuilder[String, RecosHoseMessage]()

.dest("/s/kafka/recommendations:kafka-tls")

.groupId(KafkaGroupId(f"user\_tweet\_graph-${shardId()}%06d"))

.keyDeserializer(new StringDeserializer)

.valueDeserializer(ScalaSerdes.Thrift[RecosHoseMessage].deserializer)

.seekStrategy(SeekStrategy.REWIND)

.rewindDuration(48.hours)

.withConfig(CommonClientConfigs.SECURITY\_PROTOCOL\_CONFIG, SecurityProtocol.SASL\_SSL.toString)

.withConfig(SslConfigs.SSL\_TRUSTSTORE\_LOCATION\_CONFIG, truststoreLocation())

.withConfig(SaslConfigs.SASL\_MECHANISM, SaslConfigs.GSSAPI\_MECHANISM)

.withConfig(SaslConfigs.SASL\_KERBEROS\_SERVICE\_NAME, "kafka")

.withConfig(SaslConfigs.SASL\_KERBEROS\_SERVER\_NAME, "kafka")

val graphWriter =

UserTweetGraphWriter(

shardId().toString,

serviceEnv(),

hoseName(),

128, // keep the original setting.

kafkaConfigBuilder,

clientId.name,

statsReceiver,

)

graphWriter.initHose(graph)

// For MutualTLS

val serviceIdentifier = ServiceIdentifier(

role = serviceRole(),

service = serviceName(),

environment = serviceEnv(),

zone = dataCenter()

)

log.info(s"ServiceIdentifier = ${serviceIdentifier.toString}")

val socialGraphClient: SocialGraphService.MethodPerEndpoint =

makeThriftClient[SocialGraphService.MethodPerEndpoint](

"/s/socialgraph/socialgraph",

"socialgraph",

serviceIdentifier)

val userRecentFollowersStore: ReadableStore[UserRecentFollowersStore.Query, Seq[UserId]] =

new UserRecentFollowersStore(socialGraphClient)

val tweetBasedRelatedTweetsHandler = new TweetBasedRelatedTweetsHandler(graph, statsReceiver)

val consumersBasedRelatedTweetsHandler =

new ConsumersBasedRelatedTweetsHandler(graph, statsReceiver)

val producerBasedRelatedTweetsHandler =

new ProducerBasedRelatedTweetsHandler(graph, userRecentFollowersStore, statsReceiver)

val decider = UserTweetGraphDecider(serviceEnv(), dataCenter())

val endpointLoadShedder = new EndpointLoadShedder(decider)

val userTweetGraph =

new UserTweetGraph(

tweetBasedRelatedTweetsHandler,

producerBasedRelatedTweetsHandler,

consumersBasedRelatedTweetsHandler,

endpointLoadShedder)(timer)

val thriftServer = ThriftMux.server

.withOpportunisticTls(OpportunisticTls.Required)

.withMutualTls(serviceIdentifier)

.serveIface(servicePort(), userTweetGraph)

log.info("clientid: " + clientId.toString)

log.info("servicePort: " + servicePort().toString)

log.info("adding shutdown hook")

onExit {

graphWriter.shutdown()

thriftServer.close(shutdownTimeout().fromNow)

}

log.info("added shutdown hook")

// Wait on the thriftServer so that shutdownTimeout is respected.

Await.result(thriftServer)

}

}