package com.twitter.frigate.pushservice.refresh\_handler

import com.twitter.finagle.stats.BroadcastStatsReceiver

import com.twitter.finagle.stats.Stat

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.base.Stats.track

import com.twitter.frigate.common.base.\_

import com.twitter.frigate.common.config.CommonConstants

import com.twitter.frigate.common.util.PushServiceUtil.FilteredRefreshResponseFut

import com.twitter.frigate.pushservice.model.PushTypes.PushCandidate

import com.twitter.frigate.pushservice.model.PushTypes.Target

import com.twitter.frigate.pushservice.take.CandidateNotifier

import com.twitter.frigate.pushservice.util.ResponseStatsTrackUtils.trackStatsForResponseToRequest

import com.twitter.frigate.pushservice.thriftscala.PushStatus

import com.twitter.frigate.pushservice.thriftscala.RefreshResponse

import com.twitter.util.Future

import com.twitter.util.JavaTimer

import com.twitter.util.Timer

class RefreshForPushNotifier(

rfphStatsRecorder: RFPHStatsRecorder,

candidateNotifier: CandidateNotifier

)(

globalStats: StatsReceiver) {

private implicit val statsReceiver: StatsReceiver =

globalStats.scope("RefreshForPushHandler")

private val pushStats: StatsReceiver = statsReceiver.scope("push")

private val sendLatency: StatsReceiver = statsReceiver.scope("send\_handler")

implicit private val timer: Timer = new JavaTimer(true)

private def notify(

candidatesResult: CandidateResult[PushCandidate, Result],

target: Target,

receivers: Seq[StatsReceiver]

): Future[RefreshResponse] = {

val candidate = candidatesResult.candidate

val predsResult = candidatesResult.result

if (predsResult != OK) {

val invalidResult = predsResult

invalidResult match {

case Invalid(Some(reason)) =>

Future.value(RefreshResponse(PushStatus.Filtered, Some(reason)))

case \_ =>

Future.value(RefreshResponse(PushStatus.Filtered, None))

}

} else {

rfphStatsRecorder.trackPredictionScoreStats(candidate)

val isQualityUprankingCandidate = candidate.mrQualityUprankingBoost.isDefined

val commonRecTypeStats = Seq(

statsReceiver.scope(candidate.commonRecType.toString),

globalStats.scope(candidate.commonRecType.toString)

)

val qualityUprankingStats = Seq(

statsReceiver.scope("QualityUprankingCandidates").scope(candidate.commonRecType.toString),

globalStats.scope("QualityUprankingCandidates").scope(candidate.commonRecType.toString)

)

val receiversWithRecTypeStats = {

if (isQualityUprankingCandidate) {

receivers ++ commonRecTypeStats ++ qualityUprankingStats

} else {

receivers ++ commonRecTypeStats

}

}

track(sendLatency)(candidateNotifier.notify(candidate).map { res =>

trackStatsForResponseToRequest(

candidate.commonRecType,

candidate.target,

res,

receiversWithRecTypeStats

)(globalStats)

RefreshResponse(res.status)

})

}

}

def checkResponseAndNotify(

response: Response[PushCandidate, Result],

targetUserContext: Target

): Future[RefreshResponse] = {

val receivers = Seq(statsReceiver)

val refreshResponse = response match {

case Response(OK, processedCandidates) =>

// valid rec candidates

val validCandidates = processedCandidates.filter(\_.result == OK)

// top rec candidate

validCandidates.headOption match {

case Some(candidatesResult) =>

candidatesResult.result match {

case OK =>

notify(candidatesResult, targetUserContext, receivers)

.onSuccess { nr =>

pushStats.scope("result").counter(nr.status.name).incr()

}

case \_ =>

targetUserContext.isTeamMember.flatMap { isTeamMember =>

FilteredRefreshResponseFut

}

}

case \_ =>

FilteredRefreshResponseFut

}

case Response(Invalid(reason), \_) =>

// invalid target with known reason

FilteredRefreshResponseFut.map(\_.copy(targetFilteredBy = reason))

case \_ =>

// invalid target

FilteredRefreshResponseFut

}

val bStats = BroadcastStatsReceiver(receivers)

Stat

.timeFuture(bStats.stat("latency"))(

refreshResponse

.raiseWithin(CommonConstants.maxPushRequestDuration)

)

.onFailure { exception =>

rfphStatsRecorder.refreshRequestExceptionStats(exception, bStats)

}

}

}