package com.twitter.frigate.pushservice.take

import com.twitter.conversions.DurationOps.\_

import com.twitter.finagle.stats.StatsReceiver

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

import com.twitter.frigate.common.logger.MRLogger

import com.twitter.frigate.common.store.Fail

import com.twitter.frigate.common.store.IbisResponse

import com.twitter.frigate.common.store.InvalidConfiguration

import com.twitter.frigate.common.store.NoRequest

import com.twitter.frigate.common.store.Sent

import com.twitter.frigate.common.util.CasLock

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

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

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

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

import com.twitter.frigate.pushservice.predicate.CasLockPredicate

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

import com.twitter.frigate.pushservice.take.history.\_

import com.twitter.frigate.pushservice.util.CopyUtil

import com.twitter.frigate.pushservice.thriftscala.PushResponse

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

import com.twitter.frigate.pushservice.util.OverrideNotificationUtil

import com.twitter.frigate.thriftscala.ChannelName

import com.twitter.util.Future

class CandidateNotifier(

notificationSender: NotificationSender,

casLock: CasLock,

historyWriter: HistoryWriter,

eventBusWriter: EventBusWriter,

ntabOnlyChannelSelector: NtabOnlyChannelSelector

)(

implicit statsReceiver: StatsReceiver) {

private lazy val casLockPredicate =

CasLockPredicate(casLock, expiryDuration = 10.minutes)(statsReceiver)

private val candidateNotifierStats = statsReceiver.scope(this.getClass.getSimpleName)

private val historyWriteCounter =

candidateNotifierStats.counter("simply\_notifier\_history\_write\_num")

private val loggedOutHistoryWriteCounter =

candidateNotifierStats.counter("logged\_out\_simply\_notifier\_history\_write\_num")

private val notificationSenderLatency =

candidateNotifierStats.scope("notification\_sender\_send")

private val log = MRLogger("CandidateNotifier")

private def mapIbisResponse(ibisResponse: IbisResponse): PushResponse = {

ibisResponse match {

case IbisResponse(Sent, \_) => SentResponse

case IbisResponse(Fail, \_) => SendFailedResponse

case IbisResponse(InvalidConfiguration, \_) => InvalidConfigResponse

case IbisResponse(NoRequest, \_) => NtabWriteOnlyResponse

}

}

/\*\*

\* - write to history store

\* - send the notification

\* - scribe the notification

\*

\* final modifier is to signal that this function cannot be overriden. There's some critical logic

\* in this function, and it's helpful to know that no sub-class overrides it.

\*/

final def notify(

candidate: PushCandidate,

): Future[PushResponse] = {

if (candidate.target.isDarkWrite) {

notificationSender.sendIbisDarkWrite(candidate).map(mapIbisResponse)

} else {

casLockPredicate(Seq(candidate)).flatMap { casLockResults =>

if (casLockResults.head || candidate.target.pushContext

.exists(\_.skipFilters.contains(true))) {

Future

.join(

candidate.target.isSilentPush,

OverrideNotificationUtil

.getOverrideInfo(candidate, candidateNotifierStats),

CopyUtil.getCopyFeatures(candidate, candidateNotifierStats)

).flatMap {

case (isSilentPush, overrideInfoOpt, copyFeaturesMap) =>

val channels = ntabOnlyChannelSelector.selectChannel(candidate)

channels.flatMap { channels =>

candidate

.frigateNotificationForPersistence(

channels,

isSilentPush,

overrideInfoOpt,

copyFeaturesMap.keySet).flatMap { frigateNotificationForPersistence =>

val result = if (candidate.target.isDarkWrite) {

candidateNotifierStats.counter("dark\_write").incr()

Future.Unit

} else {

historyWriteCounter.incr()

historyWriter

.writeSendToHistory(candidate, frigateNotificationForPersistence)

}

result.flatMap { \_ =>

track(notificationSenderLatency)(

notificationSender

.notify(channels, candidate)

.map { ibisResponse =>

eventBusWriter

.writeToEventBus(candidate, frigateNotificationForPersistence)

mapIbisResponse(ibisResponse)

})

}

}

}

}

} else {

candidateNotifierStats.counter("filtered\_by\_cas\_lock").incr()

Future.value(PushResponse(PushStatus.Filtered, Some(casLockPredicate.name)))

}

}

}

}

final def loggedOutNotify(

candidate: PushCandidate,

): Future[PushResponse] = {

if (candidate.target.isDarkWrite) {

notificationSender.sendIbisDarkWrite(candidate).map(mapIbisResponse)

} else {

casLockPredicate(Seq(candidate)).flatMap { casLockResults =>

if (casLockResults.head || candidate.target.pushContext

.exists(\_.skipFilters.contains(true))) {

val response = candidate.target.isSilentPush.flatMap { isSilentPush =>

candidate

.frigateNotificationForPersistence(

Seq(ChannelName.PushNtab),

isSilentPush,

None,

Set.empty).flatMap { frigateNotificationForPersistence =>

val result = if (candidate.target.isDarkWrite) {

candidateNotifierStats.counter("logged\_out\_dark\_write").incr()

Future.Unit

} else {

loggedOutHistoryWriteCounter.incr()

historyWriter.writeSendToHistory(candidate, frigateNotificationForPersistence)

}

result.flatMap { \_ =>

track(notificationSenderLatency)(

notificationSender

.loggedOutNotify(candidate)

.map { ibisResponse =>

mapIbisResponse(ibisResponse)

})

}

}

}

response

} else {

candidateNotifierStats.counter("filtered\_by\_cas\_lock").incr()

Future.value(PushResponse(PushStatus.Filtered, Some(casLockPredicate.name)))

}

}

}

}

}