package com.twitter.timelines.prediction.common.aggregates.real\_time

import com.twitter.conversions.DurationOps.\_

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.storehaus.ReplicatedReadableStore

import com.twitter.storehaus.Store

import com.twitter.timelines.clients.memcache\_common.\_

import com.twitter.timelines.util.FailOpenHandler

import com.twitter.util.Future

object ServedFeaturesMemcacheConfigBuilder {

def getTwCacheDestination(cluster: String, isProd: Boolean = false): String =

if (!isProd) {

s"/srv#/test/$cluster/cache//twemcache\_timelines\_served\_features\_cache"

} else {

s"/srv#/prod/$cluster/cache/timelines\_served\_features"

}

/\*\*

\* @cluster The DC of the cache that this client will send requests to. This

\* can be different to the DC where the summingbird job is running in.

\* @isProd Define if this client is part of a production summingbird job as

\* different accesspoints will need to be chosen.

\*/

def build(cluster: String, isProd: Boolean = false): StorehausMemcacheConfig =

StorehausMemcacheConfig(

destName = getTwCacheDestination(cluster, isProd),

keyPrefix = "",

requestTimeout = 200.milliseconds,

numTries = 2,

globalTimeout = 400.milliseconds,

tcpConnectTimeout = 200.milliseconds,

connectionAcquisitionTimeout = 200.milliseconds,

numPendingRequests = 1000,

isReadOnly = false

)

}

/\*\*

\* If lookup key does not exist locally, make a call to the replicated store(s).

\* If value exists remotely, write the first returned value to the local store

\* and return it. Map any exceptions to None so that the subsequent operations

\* may proceed.

\*/

class LocallyReplicatedStore[-K, V](

localStore: Store[K, V],

remoteStore: ReplicatedReadableStore[K, V],

scopedStatsReceiver: StatsReceiver)

extends Store[K, V] {

private[this] val failOpenHandler = new FailOpenHandler(scopedStatsReceiver.scope("failOpen"))

private[this] val localFailsCounter = scopedStatsReceiver.counter("localFails")

private[this] val localWritesCounter = scopedStatsReceiver.counter("localWrites")

private[this] val remoteFailsCounter = scopedStatsReceiver.counter("remoteFails")

override def get(k: K): Future[Option[V]] =

failOpenHandler {

localStore

.get(k)

.flatMap {

case Some(v) => Future.value(Some(v))

case \_ => {

localFailsCounter.incr()

val replicatedOptFu = remoteStore.get(k)

// async write if result is not empty

replicatedOptFu.onSuccess {

case Some(v) => {

localWritesCounter.incr()

localStore.put((k, Some(v)))

}

case \_ => {

remoteFailsCounter.incr()

Unit

}

}

replicatedOptFu

}

}

} { \_: Throwable => Future.None }

}