package com.twitter.servo.util

import com.twitter.util.Future

object RpcRetry {

/\*\*

\* Provides a generic implementation of a retry logic to only a subset

\* of requests according to a given predicate and returning the result

\* in the original order after the retry.

\* @param rpcs Methods that can transform a Seq[Request] to

\* Future[Map[Request, Response]], they will be invoked in order

\* while there are remaining rpcs to invoke AND some responses

\* still return false to the predicate.

\* @param isSuccess if true, keep the response, else retry.

\* @tparam Req a request object

\* @tparam Resp a response object

\* @return an rpc function (Seq[Req] => Future[Map[Req, Resp]]) that performs

\* the retries internally.

\*/

def retryableRpc[Req, Resp](

rpcs: Seq[Seq[Req] => Future[Map[Req, Resp]]],

isSuccess: Resp => Boolean

): Seq[Req] => Future[Map[Req, Resp]] = {

requestRetryAndMerge[Req, Resp](\_, isSuccess, rpcs.toStream)

}

/\*\*

\* Provides a generic implementation of a retry logic to only a subset

\* of requests according to a given predicate and returning the result

\* in the original order after the retry.

\* @param rpcs Methods that can transform a Seq[Request] to

\* Future[Seq[Response]], they will be invoked in order

\* while there are remaining rpcs to invoke AND some responses

\* still return false to the predicate.

\* Note that all Request objects must adhere to hashCode/equals standards

\* @param isSuccess if true, keep the response, else retry.

\* @tparam Req a request object. Must adhere to hashCode/equals standards

\* @tparam Resp a response object

\* @return an rpc function (Seq[Req] => Future[Seq[Resp]]) that performs

\* the retries internally.

\*/

def retryableRpcSeq[Req, Resp](

rpcs: Seq[Seq[Req] => Future[Seq[Resp]]],

isSuccess: Resp => Boolean

): Seq[Req] => Future[Seq[Resp]] = {

requestRetryAndMergeSeq[Req, Resp](\_, isSuccess, rpcs)

}

private[this] def requestRetryAndMergeSeq[Req, Resp](

requests: Seq[Req],

isSuccess: Resp => Boolean,

rpcs: Seq[Seq[Req] => Future[Seq[Resp]]]

): Future[Seq[Resp]] = {

requestRetryAndMerge(requests, isSuccess, (rpcs map { rpcToMapResponse(\_) }).toStream) map {

responseMap =>

requests map { responseMap(\_) }

}

}

private[this] def requestRetryAndMerge[Req, Resp](

requests: Seq[Req],

isSuccess: Resp => Boolean,

rpcs: Stream[Seq[Req] => Future[Map[Req, Resp]]]

): Future[Map[Req, Resp]] = {

if (rpcs.isEmpty) {

Future.exception(new IllegalArgumentException("rpcs is empty."))

} else {

val rpc = rpcs.head

rpc(requests) flatMap { responses =>

val (keep, recurse) = responses partition {

case (\_, rep) => isSuccess(rep)

}

if (rpcs.tail.nonEmpty && recurse.nonEmpty) {

requestRetryAndMerge(recurse.keys.toSeq, isSuccess, rpcs.tail) map { keep ++ \_ }

} else {

Future.value(responses)

}

}

}

}

private[this] def rpcToMapResponse[Req, Resp](

rpc: Seq[Req] => Future[Seq[Resp]]

): Seq[Req] => Future[Map[Req, Resp]] = { (reqs: Seq[Req]) =>

rpc(reqs) map { reps =>

(reqs zip reps).toMap

}

}

}